Vision Egg: an open-source library for realtime visual stimulus generation

Andrew D. Straw*

Bioengineering, California Institute of Technology, Pasadena, CA, USA

Edited by: Rolf Kötter, Radboud University Nijmegen, Netherlands Antilles

Reviewed by: Laurent Perrinet, INCM-CNRS, France
Jonathan W. Peirce, University of Nottingham, UK

Modern computer hardware makes it possible to produce visual stimuli in ways not previously possible. Arbitrary scenes, from traditional sinusoidal gratings to naturalistic 3D scenes can now be specified on a frame-by-frame basis in realtime. A programming library called the Vision Egg that aims to make it easy to take advantage of these innovations. The Vision Egg is a free, open-source library making use of OpenGL and written in the high-level language Python with extensions in C. Careful attention has been paid to the issues of luminance and temporal calibration, and several interfacing techniques to input devices such as mice, movement tracking systems, and digital triggers are discussed. Together, these make the Vision Egg suitable for many psychophysical, electrophysiological, and behavioral experiments. This software is available for free download at visionegg.org.

Keywords: visual stimulus generation, open source, Python

INTRODUCTION

A neuroscientist may need precisely defined spatial, temporal, spectral, and polarization properties of light to perform a particular visual experiment. Standard computer monitors and projectors are capable of producing a wide range of stimuli sufficient for many experiments, and special purpose displays may be built or purchased with a standard interface. A tool which produces precisely controlled signals from a video port (such as VGA) is therefore of great utility. This paper outlines the Vision Egg, a programming library developed to serve as such a tool in combination with a standard computer and other software libraries.

HISTORICAL CONTEXT

A brief outline of the display systems with the most impact on the design of the Vision Egg follows.

In the 1980s and 1990s, vision scientists frequently displayed their stimuli on a Tec-tronix 608 display, a small (~12 cm diagonal) cathode ray tube with independent X,Y and luminance inputs originally intended for use in a high-bandwidth analog oscilloscope. However, instead of using it as an oscilloscope display, vision scientists often controlled the 608 with an Innisfree Picasso device, a specialized function generator that creates a raster scan of X,Y positions and modulates luminance to produce a variety of simple stimuli such as sinusoidal gratings and rectangles. Many scientists found the Picasso wonderfully easy to use, as its intuitive interface with a myriad of switches and potentiometers allowed rapid experimentation until a suitable stimulus was found. Furthermore, by providing BNC connections for voltage inputs, time-varying stimuli could be driven via analog outputs from the same data acquisition system being used to record responses, simplifying experimental design. The main limitations of the Picasso are essential to its design as a specialized function generator – namely that it is tied to a specific (and now rare) display device, and that the range of stimuli it could produce were limited.

Computers provide the ability to produce arbitrary visual stimuli, but with a new set of limitations. Early systems developed in the 1990s required no specialized hardware but could only draw pre-rendered stimuli and movies (e.g., early releases of the PsychToolbox: Brainard, 1997; Pelli, 1997) or were limited to simple stimuli and required extensive programming and debugging in low-level C (e.g., John Maunsell's custom LabLib). These systems achieved frame-by-frame temporal precision by operating within a cooperative multitasking operating system such as Mac OS (prior to Mac OS X) and running at interrupt time. Under such conditions, the underlying OS would not preempt a program's use of the CPU or other resources. With the rise of pre-emptive multitasking operating systems such as Windows 95, GNU/Linux, and Mac OS X, such an approach to precise timing was no longer guaranteed. Another issue, which persists today, is that the general-purpose nature of display hardware meant that producing stimuli with a large dynamic range of contrast can be difficult.

Custom hardware solutions, such as the Cambridge Research Systems' VSG 2/3F, addressed the issues of precise timing and

*Correspondence: Andrew D. Straw, Bioengineering, California Institute of Technology, 1200 E. California Boulevard, Mail Code 138-78, Pasadena, CA 91125, USA. e-mail: astraw@caltech.edu

Received: 15 September 2008; paper pending published: 26 September 2008; accepted: 08 October 2008; published online: 04 November 2008.

Citation: *Front. Neuroinform.* (2008) **2**: 4. doi: 10.3389/neuro.11.004.2008

dynamic range through the use of special purpose processing units and digital to analog converters isolated from the main computer system on a PCI card. Programs would execute onboard these cards independently from the host operating system, bypassing the issues outlined above. Such cards were expensive, however, often costing five or more times the price of the host computer itself, with additional RAM costing still more. Additionally, programming the VSG 2/3F involved either using a script language with limited performance or a low-level, assembly-like language specific to the processing unit onboard the card.

By the year 2000, OpenGL, a library to abstract standard graphics hardware, was being used for realtime generation of 3D graphics on broadcast television without skipping frames. I was encouraged to try a similar approach for my own experiments on the visual system of flies, where the ability to use 3D video acceleration hardware was appealing because it meant that wide-field stimuli could be accurate across displays subtending very large angles. Such graphics hardware was appealing more generally for vision research because this hardware was very fast at mathematical operations involved in drawing scenes while the open nature of the OpenGL specification meant that solutions would be portable to future hardware. The high speed allowed new possibilities for the display of visual stimuli that change over time. Dynamic scenes of high complexity, including in 3D, could be rendered in realtime, only an instant before display. This could be done at high update rates without skipping frames, and these video cards could display anything from simple shapes to naturalistic 3D scenes. The immediate benefit for my research was to enable drawing at 200 Hz of perspective-corrected Gabor wavelets (Straw et al., 2006) and temporally anti-aliased (so called *motion blurred)* moving natural images (Straw et al., 2008). Both of these types of stimuli had been very difficult to implement with the other systems.

OPEN SOURCE SOFTWARE AND PYTHON

Fundamental to the scientific process is the repeatability of measurements. For this reason, open source software should be preferred in scientific applications – this prevents software mistakes from becoming hidden in proprietary code, allows others to learn from and independently reproduce work, and allows a community approach to solve problems together. As illustrated by the articles in this issue, Python is becoming a standard high-level, open source language in neuroscience. Perhaps the most exciting aspect of the confluence of tools available in Python is the possibility of software that incorporates components from various sources into software with new capabilities. The suitability of Python for drawing visual stimuli is well described in Peirce (2007), and additional notes are in Section "Timing of Visual Stimuli: Speed and Latency." The Vision Egg also makes use of software for which no Python interface previously existed. These function calls are written as C extension modules to Python included with the Vision Egg.

VISION EGG

The aim of this paper is to describe the Vision Egg, an open source (LPGL license) computer programming library which makes use of modern hardware accelerated graphics using OpenGL to generate visual stimuli. One important goal for the project is to allow non-experts to use modern computer hardware to its maximum capability for common vision science tasks. A screenshot of an included demonstration script showing several of the visual stimulus possibilities is shown in **Figure 1**, and source code to a moving sinusoidal grating is shown in **Figure 2**.

At the initial development and release of the Vision Egg in 2001–2002, existing software for vision scientists was not able to take advantage of the capabilities present in the emerging hardware standards. Now, almost every personal computer being sold is equipped with graphics hardware suitable for many

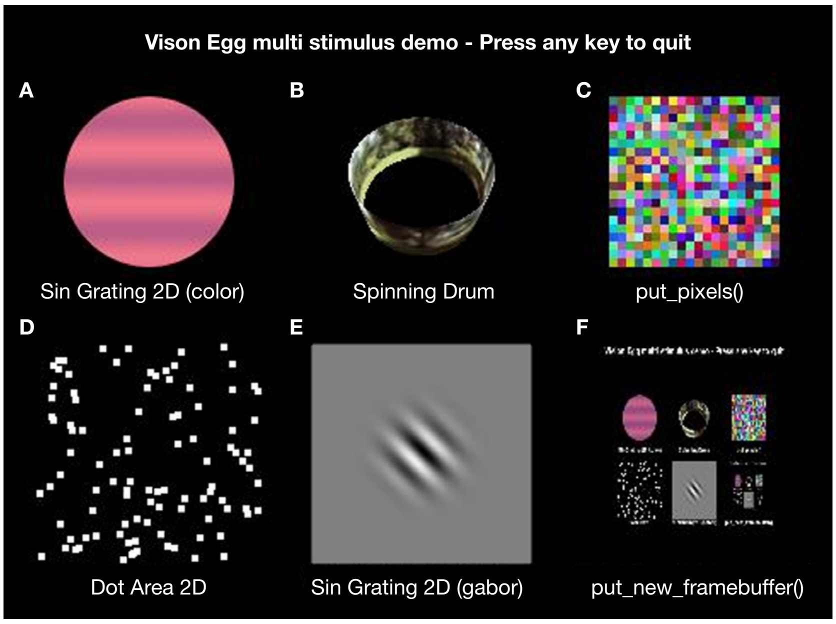


Figure 1 | **Screenshot of Vision Egg `multi_stim.py` demonstration script showing several included visual stimulus types.** The dynamic stimuli are updated in realtime without skipping frames at rates up to the fastest vertical refresh rate of the display tested (200 Hz). Stimuli, are: ***(A)*** A circularly windowed color grating changing in space and color over time. ***(B)*** A rotating, perspective distorted drum with a natural panorama used as a texture image. ***(C)*** Arbitrary arrays of RGB data updated on each frame generated from a uniform random distribution. ***(D)*** Random dot stimuli with 100 independently moving dots. ***(E)*** A drifting Gaussian windowed sinusoidal grating. ***(F)*** A copy of the framebuffer recursively redrawn at smaller scale.

A
```
#  Import modules
import VisionEgg
VisionEgg.start_default_logging(); VisionEgg.watch_exceptions()

from VisionEgg.Core import get_default_screen, Viewport
from VisionEgg.FlowControl import Presentation
from VisionEgg.Gratings import SinGrating2D

#  Initialize OpenGL window/screen
screen = get_default_screen()

#  Create sinusoidal grating object
stimulus = SinGrating2D(spatial_freq     = 10.0 / screen.size[0],
                        temporal_freq_hz = 1.0,
                        )

#  Create viewport - intermediary between stimuli and screen
viewport = Viewport( screen=screen, stimuli=[stimulus] )

#  Create presentation object and go
p = Presentation(go_duration=(5.0,'seconds'),viewports=[viewport])
p.go()
```

B
```
#  Import modules
import VisionEgg
VisionEgg.start_default_logging(); VisionEgg.watch_exceptions()

from VisionEgg.Core import get_default_screen, Viewport, \
    swap_buffers
from VisionEgg.Gratings import SinGrating2D

#  Initialize OpenGL window/screen
screen = get_default_screen()

#  Create sinusoidal grating object
stimulus = SinGrating2D(spatial_freq     = 10.0 / screen.size[0]
                        temporal_freq_hz = 1.0,
                        )

#  Create viewport - intermediary between stimuli and screen
viewport = Viewport( screen=screen, stimuli=[stimulus] )

# Use our own main loop
tstart = VisionEgg.time_func()
while (VisionEgg.time_func() - tstart) <= 5.0:
    screen.clear()
    viewport.draw()
    swap_buffers()
```

Figure 2 | **Source code of simple Vision Egg program to draw a moving sinusoidal grating illustrating a simple but complete program.** Two means of controlling the flow of execution are available, as described in Section "Mid-level Software Overview: Controlling Program Flow." ***(A)*** Program flow is controlled by the Vision Egg's `Presentation` class. ***(B)*** Program flow is explicitly specified within the script.

experiments. Although more expensive hardware, often designed with computer games in mind, continues to push the limits of performance, the modest graphics systems now found in laptops and some motherboards perform fine for many experimental purposes. Even the creation of artificially closed-loop "virtual-reality" experiments with the Vision Egg is possible with relatively inexpensive hardware (e.g., Fry et al., 2004, 2008) but the library is also useful for a variety of simpler tasks.

The biggest challenge with such an approach is addressing potential problems when attempting to produce precisely controlled stimuli for visual science on hardware which was not explicitly designed for the task. The remainder of this paper describes the implementation of the Vision Egg, some experiments to characterize its performance, a discussion of it in relation to other visual stimulus technologies, and some potential future directions.

LOW-LEVEL HARDWARE AND SOFTWARE OVERVIEW

HARDWARE

This section presents a brief review of modern computer architecture from a hardware perspective for drawing visual stimuli. Applications run on the CPU of the host computer, though which they manipulate the memory, video system, and other devices of the computer. Video cards have onboard graphics processors (GPUs) that are faster than CPUs at pushing pixels. By shifting the majority of the drawing work onto the video card, the role of the CPU can be limited to directing the powerful GPU. To render a complicated 3D scene, for example, the CPU computes a wireframe model that is transmitted, along with rasterization instructions such as texture images and coordinates, to the video card. This communication is specified by OpenGL, which hides the hardware level details such as transmission of data across the computer bus. The GPU renders this image to a framebuffer, which is then read out either by a high-speed digital to analog converter (RAMDAC) or a digital transmitter (e.g., DVI, HDMI, and Display Port). Luminance and color information is limited in typical framebuffers because they store 8 bits per color per pixel, or 256^3 values of red, green, and blue each for a total of 256^3 (16.6 million) possible colors. The RAMDAC converts these digital values to an analog voltage after passing them through a color lookup table, which can be used to correct non-linearities the display process such as *gamma* (see Section "Precise Control of Color and Luminance: Results of Luminance Calibration"). Recently, manufacturers have been increasing the precision of the lookup tables in the RAMDAC, and although many 8 bit per color RAMDACs are still available, 10 bit cards are becoming more common. Furthermore, some higher-end cards have 10 bit framebuffers.

DRAWING IN OpenGL

The Vision Egg scripts enter a loop which draws a new frame on each cycle. Often each frame can be drawn completely from scratch, allowing realtime control of stimuli or simply to eliminate a common brute force approach of pre-rendering several frames and then displaying them sequentially. Furthermore, the frame skips do not lead to cumulative error if each frame is drawn in realtime based on an accurate clock time. In an OpenGL system, a double buffering technique is used, meaning that new frames are rendered to the back framebuffer while the RAMDAC draws the contents of the front buffer to the display. Due to this double buffering, partially completed frames are not drawn to the screen. When finished rendering to the back framebuffer, the application informs the graphics system to use the back buffer as the source of data for the RAMDAC. Thus, the front and back buffers are swapped (with an OpenGL `flip()` or Vision Egg `swap_buffers()` function call) and drawing continues on the new back buffer. In the so-called *vsync* (vertical sync) mode, the buffer swap is synchronized to occur only between frame draws by the display, and thus no "tearing" artifacts are present. With small displacements between individual frames, however, tearing is minimal without using vertical sync. Regardless of vsync mode, the main loop OpenGL delays execution of the program until the buffer swap command is sent to the video hardware.

A member of the Vision Egg community has performed extensive testing on the latencies associated with drawing in OpenGL (Sol Simpson, SR Research, personal communication), which are in agreement with my personal observations and more limited testing. His tests show that even with vsync on, the actual call to `swap_buffers()` acts in an asynchronous manner when no buffer swaps are pending, but begins blocking when another swap is scheduled. In other words, the first call to `swap_buffers()` will return immediately and the graphics card is instructed to swap buffers during the next vertical retrace. However, if another call to `swap_buffers()` is issued before the retrace occurs, this call is blocked (does not return) until the first scheduled buffer swap happens. Thus, a program which paces itself via returning from blocked calls to `swap_buffers()` will always be drawing frames which will be drawn not on the next buffer swap, but on the second buffer swap.

Thus, if a program calls `swap_buffers()` less than once per retrace interval, then the `swap_buffers()` call is not blocked and returns right away and not necessarily at the start of a retrace. In this case, one does not see a constant 1 retrace interval delay. Instead, one will see a variable delay (the time between when `swap_buffers()` returns and when the display is actually updated), with a duration up to the retrace interval depending on when `swap_buffers()` was called.

This suggests that one cannot not rely on when `swap_buffers()` returns to determine when the flip actually occurs and instead should use a combination of `swap_buffers()` followed by some code that actually waits until, or determines, the start of the next retrace. The Vision Egg currently provides such a function for Windows (see Section "Low-level Hardware and Software Overview: Detecting Retrace Events and Refresh Rates"). The same results are found with the Vision Egg, pure C OpenGL and with SDL when using the DirectX backend on ATI and nVIDIA graphics cards (Sol Simpson, SR Research, personal communication).

Due to the intricacies of the above latency issue when vsync is on and the lack of a way to detect retrace events on all supported platforms, the Vision Egg currently (up to and including 1.1.1) simply assumes that frames are drawn when `swap_buffers()` returns. This gives an accurate estimate of whether refresh intervals were skipped and consequently a frame was not updated, but results in latency increased by one refresh interval.

Recent video cards (e.g., nVIDIA GeForce 8500 GT with the Forceware version 163.71 driver on Windows XP) support "triple buffering." In this mode, there are two back buffers that are alternately drawn upon, and the most recently completed buffer is used at the start of display of a new frame to the screen. Although I have not tested this technique, it theoretically allows near-minimal latencies without tearing artifacts or difficult programming involving refresh detection.

OPERATING SYSTEMS

The Vision Egg runs on any platform which supports Python and OpenGL. It is known to run on Microsoft Windows (95, 2000, and XP), GNU/Linux with kernels 2.4 and 2.6 (Ubuntu, Redhat, Debian), Mac OS X and SGI IRIX. All of these are preemptive multitasking operating systems, with important ramifications described in section "Timing of Visual Stimuli: Speed and Latency."

DETECTING RETRACE EVENTS AND REFRESH RATES

The Vision Egg offers some platform-dependent features. One of these is the ability to detect or wait for a vertical retrace event. This is implemented according to the method of Riemersma (2000) and implemented in the `Win32_vretrace.pyx` file. Furthermore, the refresh rate can be detected on Windows and Mac OS X as implemented in the `win32_getrefresh.c` and `darwin_getrefresh.m` files. Unfortunately, the Vision Egg does not currently allow the user to set the refresh rate.

MAXIMUM PRIORITY MODE

Operating systems typically have means to boost the priority of some processes above that of other processes. The details are specific to each platform, but the Vision Egg includes support for raising priority on Windows via the `SetPriorityClass()` and `SetThreadPriority()` functions, on POSIX systems (such as Linux) via the `sched_setscheduler()` and `mlockall()` functions, and on Mac OS X via the `thread_policy_set()`, `setpriority()` and `pthread_setschedparam()` functions. On Mac OS X, these function calls tell the kernel's realtime scheduler to grant programs a periodic time slice from the CPU, which theoretically might give hard realtime performance (guaranteed latency), but practically is limited by the issues described in Section "Timing of Visual Stimuli: Speed and Latency."

MID-LEVEL SOFTWARE OVERVIEW

DISPLAY OF STIMULI

The Vision Egg has methods to draw a wide variety of stimulus types. These stimuli operate within defined guidelines so that they only modify certain values of the OpenGL state machine, but leave all other values unchanged. In this way, multiple stimuli can be combined simultaneously, as in **Figure 1**. Both 2D and 3D stimuli are available. 2D stimuli commonly use an orthographic projection such that coordinates are specified in pixel units. Perspective projections can be used for 3D stimuli such that a calibrated projection will provide an accurate representation of object shapes when viewed on a flat display (e.g., Kern et al., 2001; Straw et al., 2006). Included with the Vision Egg are routines for drawing luminance sinusoidal gratings (2D or 3D, with or without contrast windows, which can be circular or anisotropic Gaussian in shape), color sinusoidal gratings, random dot stimuli, arbitrary image files, arbitrary numeric array data, QuickTime movies, MPEG movies, a spinning 3D drum with a textured image, rectangles and fixation points.

Many features of OpenGL are supported, including realtime resampling of the texture image data using linear interpolation and use of mipmapped textures generated with bicubic interpolation (or other means). These features allow display of slowly moving images without quantization of other systems where pixel-by-pixel steps must be made in integer multiples of the inter-frame interval. Other features, such as realtime lighting and shadows, are not currently implemented.

USER INTERACTION AND ALTERNATIVE SOURCES OF INPUT

User interaction, such as handling of keystrokes, mouse clicks, and joysticks can occur within the main loop of a Vision Egg program by using the pygame library. Additionally, because the Vision Egg is written in Python and can be easily extended with C, there are many potential sources of external input. For example, the UDP network protocol is frequently used in online computer games for low latency network communication and can be used for realtime control of visual stimuli from an external program. In this manner, a Vision Egg script may be written which is controlled from a data acquisition environment written in Python, Lab View, or MATLAB. The TCP network protocol, although slower than UDP, offers built-in error checking and correction, and has been used to provide realtime input for the Vision Egg (Fry et al., 2004, 2008).

CONTROLLING PROGRAM FLOW

The Vision Egg offers two ways of program flow control. The most conceptually simple of these is to let the programmer specify what happens on every frame, as illustrated in Figure 2B.

Because the Vision Egg was originally developed for studies in which controlling motion adaptation was critical, I paid careful attention to issues such as allowing a stimulus to continue moving while not in an experimental trial. The result is the programmer relinquishes control by entering the `go()` method of the `Presentation` class, as defined in the `VisionEgg.FlowControl` module, as in Figure 2A. This is the concept of a *go loop*, which usually corresponds to the experimental trial, and the concept of refreshing stimuli *between go loops*. Any function calls or stimulus updates not automatically performed by the Vision Egg must be implemented by means of `Controllers`, which are implementations of callback functions. Such a *main-loop-and-callback* style of programming is common in GUI programming. For example, the WX Widgets toolkit and the Mac OS X Cocoa libraries operate this way.

HIGH-LEVEL SOFTWARE OVERVIEW

SPECIFYING GRAPHICS STATE

A configuration GUI (Figure 3) can optionally be called at the beginning of any Vision Egg script. Although all options are available from the programmatic interface, it is often convenient to see and edit these parameters through this interface. Particularly important are the options for loading the color lookup tables to perform gamma correction as illustrated in Section "Precise Control of Color and Luminance: Results of Luminance Calibration."

AN APPLICATION FOR ELECTROPHYSIOLOGY

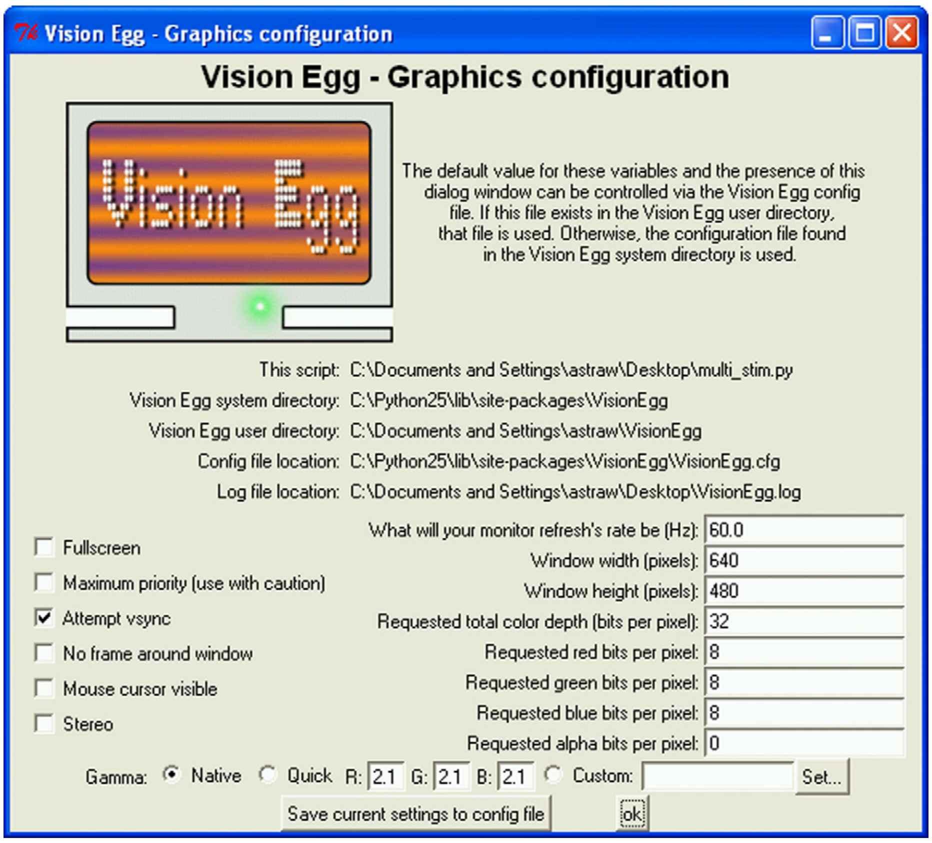
The Vision Egg includes two applications for integration within an electrophysiology environment (see Figure 4). The first is `ephys_server.py`, which draws stimuli on its video hardware. To minimize the possibility of frame skipping, this program may run as the sole application on a dedicated stimulus computer. This server program listens on a network port for a connection from the `ephys_gui.pyw` program, which offers a GUI for the experimenter to control.

THE QUEST ALGORITHM

A pure Python implementation of Watson and Pelli's (1983) QUEST algorithm is available from the Vision Egg website. This well-known Bayesian adaptive method allows estimating psychometric thresholds, and was translated directly from the MATLAB code of Denis G. Pelli, who graciously allowed redistribution of the Python version under an open-source BSD license.

QuickTime AND MPEG MOVIES

The Vision Egg includes support to decode movies and send them to OpenGL by using Apple's QuickTime API on Windows and Mac OS X and py game/SDL's `Movie` objects on all supported operating systems.



Vision Egg - Graphics configuration

The default value for these variables and the presence of this dialog window can be controlled via the Vision Egg config file. If this file exists in the Vision Egg user directory, that file is used. Otherwise, the configuration file found in the Vision Egg system directory is used.

This script:	C:\Documents and Settings\astraw\Desktop\multi_stim.py
Vision Egg system directory:	C:\Python25\lib\site-packages\VisionEgg
Vision Egg user directory:	C:\Documents and Settings\astraw\VisionEgg
Config file location:	C:\Python25\lib\site-packages\VisionEgg\VisionEgg.cfg
Log file location:	C:\Documents and Settings\astraw\Desktop\VisionEgg.log

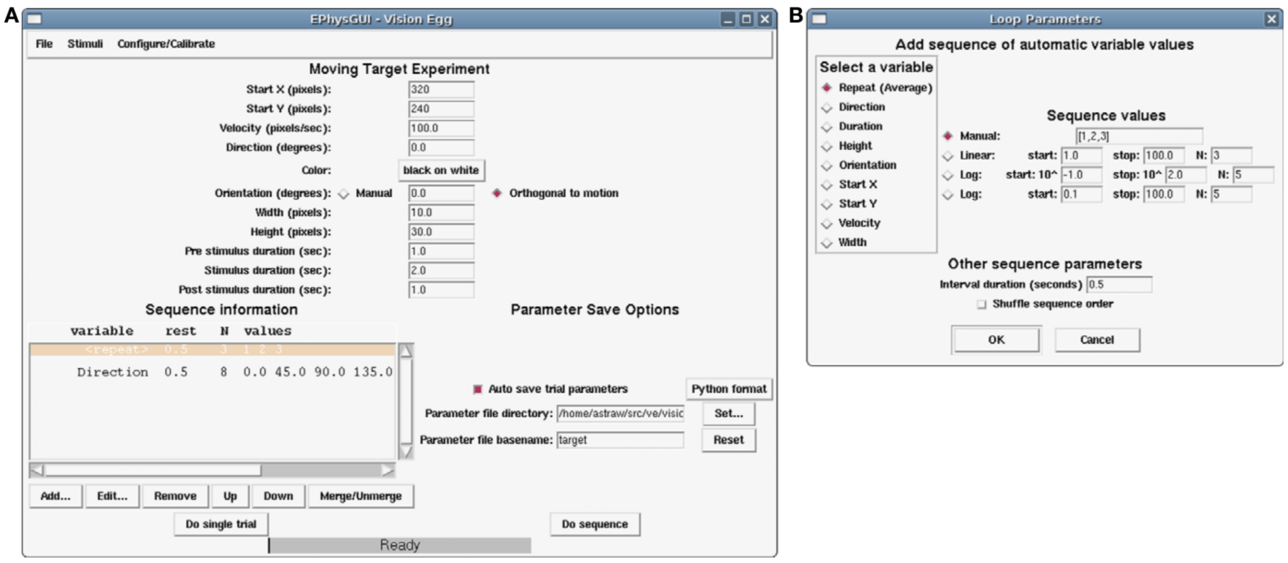
- [ ] Fullscreen
- [ ] Maximum priority (use with caution)
- [x] Attempt vsync
- [ ] No frame around window
- [ ] Mouse cursor visible
- [ ] Stereo

What will your monitor refresh's rate be (Hz):	60.0
Window width (pixels):	640
Window height (pixels):	480
Requested total color depth (bits per pixel):	32
Requested red bits per pixel:	8
Requested green bits per pixel:	8
Requested blue bits per pixel:	8
Requested alpha bits per pixel:	0

Gamma: [✓ Native] Quick R: 2.1 G: 2.1 B: 2.1 Custom: Set...

Save current settings to config file ok

Figure 3 | **Screenshot of the standard Vision Egg configuration GUI.** Numerous options for configuration are available, including framebuffer size and bit depth, color lookup tables for gamma correction and platform-dependent realtime priority, as described in Section "High-level Software Overview: Specifying Graphics State."



Moving Target Experiment

Start X (pixels):	320
Start Y (pixels):	240
Velocity (pixels/sec):	100.0
Direction (degrees):	0.0
Color:	black on white
Orientation (degrees): Manual	0.0 [✓ Orthogonal to motion]
Width (pixels):	10.0
Height (pixels):	30.0
Pre stimulus duration (sec):	1.0
Stimulus duration (sec):	2.0
Post stimulus duration (sec):	1.0

Sequence information

```
variable    rest   N  values
 <repeat>   0.5    3  1 2 3
  Direction  0.5    8  0.0 45.0 90.0 135.0
```

Parameter Save Options

[✓ Auto save trial parameters]

Parameter file directory: /home/astraw/src/ve/visic

Parameter file basename: target

Add sequence of automatic variable values

Select a variable

[✓ Repeat (Average)]
Direction
Duration
Height
Orientation
Start X
Start Y
Velocity
Width

Sequence values

[✓ Manual:]	[1,2,3]		
Linear:	start: 1.0	stop: 100.0	N: 3
Log:	start: 10^ -1.0	stop: 10^ 2.0	N: 5
Log:	start: 0.1	stop: 100.0	N: 5

Other sequence parameters

Interval duration (seconds) 0.5

Shuffle sequence order

Figure 4 | **Screenshot of electrophysiology-oriented GUI application included with the Vision Egg called** `ephys.gui.pyw` **and described in Section "High-level Software Overview: An Application for Electrophysiology."** ***(A)*** Main window shows parameters for repeated presentations of a stimulus with the possibility of automatically sequencing over variables. All settings can be saved and loaded from disk. ***(B)*** The loop parameters window allows control of experiments.

TIMING OF VISUAL STIMULI

METHODS TO MEASURE LATENCY

This section contains the results of experiments in which the total latency of the system, from input to output, was measured. Because it is difficult to measure the precise time of events happening inside and outside a computer on the same clock (or synchronized clocks), a task was chosen in which only a single time reference was necessary. The task was to measure the duration for a USB mouse movement to be translated into the movement of a rectangle drawn on the screen, both of which were filmed with a high speed video camera and later analyzed. The latencies measured in this task should be comparable to the latencies of other input–output tasks.

An LED was rigidly fixed to each computer mouse (Logitech MX-300 USB and Dell DEL1 Optical USB). The mouse was connected to a USB port on the motherboard of the computer (Acer Aspire T690 with Intel ICH7 chipset including USB2 EHCI and USB UHCI controllers). A PCI-Express xl6 video card (nVIDIA GeForce 8500 GT) was connected to a CRT monitor (Iiyama Vision Master 450) using a VGA cable. The display was set to a resolution of 800 × 600 at 140 Hz update rate using the nVIDIA control panel (Forceware version 163.71) on Windows XP Service Pack 2 and confirmed by using the monitor's on screen display.

The Vision Egg version 1.1.1 was used to draw a 3 × 3 pixel white square using the `Target2D` class on a `Screen` with a black background color in a way that it acted as a mouse cursor. The position of the mouse controlled the position of this small square using a version of the `mouseTarget.py` demo program that was simplified to remove the code that set the orientation of the target.

A high speed digital video camera (Photron Fastcam APX 120) was placed to record the LED and target location on the screen in the same image frame. Images were acquired at 2000 frames per second while the mouse was rapidly moved back and forth by hand in a roughly sinusoidal manner (e.g., Figure 5). Digital images were analyzed to identify the "center of mass" of the bright areas using the `center_of_mass()` function of the `scipy.ndimage` module. For the on-screen target, this only occurred approximately every 14th frame due to the discrete nature of raster scan CRT displays.

SPEED AND LATENCY

Because Python is an interpreted language, programs written in it will run more slowly than a well-written C program. However, Python is fast enough for two primary reasons. First, the most computation-intensive task, manipulation of large data arrays is performed with high-performance C and FORTRAN code via the numpy module of Python. Thus, Python code directs computationally intensive tasks without performing them in the slower interpreted environment. Second, computer displays cannot be refreshed beyond their maximum vertical frequency, which typically ranges up to 200 Hz. This therefore represents an upper bound on the amount of computation required for realtime rendering tasks.

In fact, the biggest timing-related concern is unrelated to the programming language used. A pre-emptive multitasking operating system may take control of the CPU from the stimulus generating program for periods longer than an inter-frame interval, thus leading to skipped frames. Even if the OS takes control of the CPU from an application for much less than an inter-frame interval, frames may still be skipped if the stimulus generation program uses a strategy of waiting until the last instant to render a frame and CPU control is taken at this critical instant. Operating systems may have some means addressing this issue such as a realtime scheduler that guarantees uninterrupted CPU time at specified intervals. The Vision Egg makes use of such facilities where available (see Section "Low-level Hardware and Software Overview: Maximum Priority Mode"). Although they can certainly help eliminate timing issues, such priority-boosting solutions cannot provide absolute guarantees about timing because OpenGL implementations themselves may be subject to unpredictable

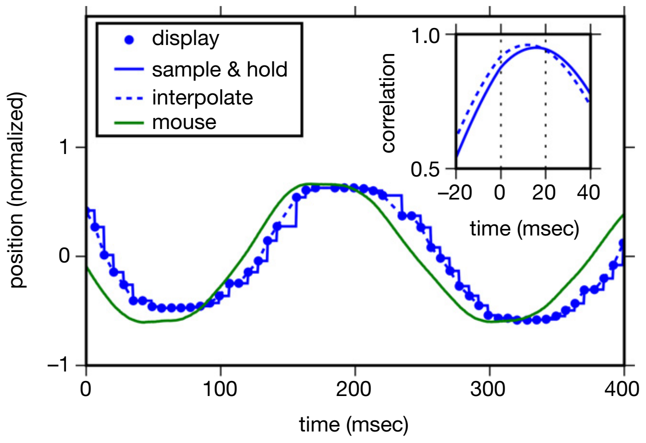


Figure 5 | **Total latency of system, including input from an optical USB mouse and display on 140 Hz CRT display, can be reduced to about 15 ms, as described in Section "Timing of Visual Stimuli: Measurements of Latency."** The main panel shows representative data gathered from a high speed camera of an LED fixed to a mouse (green line) and a bright spot on the screen controlled by the mouse (blue dots). Display positions could reasonably be interpolated using a sample-and-hold function (blue solid line) or linear interpolation (blue dashed line). Inset panel shows cross correlation of 2 s of such data when interpolated. These data were gathered with vsync off and a Logitech MX-300 USB mouse.

behavior and generally are not written to operate in a *hard realtime* (in other words, with deterministic latency) manner. For example, drawing a single additional object may cause the hardware to pass a critical threshold for memory use and force a slow operation. A low-level solution which operated in hard realtime would have to bypass complex OpenGL libraries and implement routines to draw directly to the framebuffer to guarantee performance.

It is worth noting that because of this unavoidable variable latency listed above (pre-emptive multitasking operating systems and OpenGL implementations), the variable latency introduced by use of an interpreted language with garbage collection, such as Python, does not fundamentally worsen the situation. In other words, use of Python introduces no fundamental problem other than that of an additional potential source of variable latency to that already imposed by the OS and OpenGL.

MEASUREMENTS OF LATENCY

A high speed video camera was used to measure the absolute total latency between input (a standard USB computer mouse) and output (the position of a rectangle on the screen), as described in the methods Section "Timing of Visual Stimuli: Methods to Measure Latency." **Figure 5** shows that latency can be reduced to around 15 ms, but that the vsync state plays a very significant role in total latency (**Table 1**). On a 140-Hz display (7.1 ms inter-frame interval), latency jumped by 17 ms or more when vsync was enabled. This is presumably due to the latency imposed by drawing in the middle of a refresh interval and waiting for that interval to be done combined with the additional latency described in Section "Low-level Hardware and Software Overview: Drawing in OpenGL." Although the results would be interesting, these experiments were not repeated in triple buffering mode.

ONLINE DETECTION OF FRAME SKIPPING

Frame skipping is determined by measuring the interval between successive buffer swap commands using standard system calls to query the computers clock. If this value exceeds the known monitor inter-frame interval, a frame has been skipped. Stimuli generated by the Vision Egg are routinely presented for hours without skipping a frame when measured this way. The most likely occurrence of a skipped frame is at the immediate beginning of drawing a stimulus – presumably when some initialization occurs with the video system. Often this can be dealt with by initializing the video system in a non-critical task, such as drawing a black rectangle.

Table 1 | **Latency as estimated by the peak of the cross correlation between mouse location and displayed point location.** Optimistic latencies were estimated using the cross correlation with the linearly interpolated display positions as plotted in Figure 5 and described in Section "Timing of Visual Stimuli: Measurements of Latency." Pessimistic latencies were also estimated with a cross correlation, but used a sample-and-hold function rather than linear interpolation to estimate display position.

Vsync	Mouse	Optimistic latency (ms)	Pessimistic latency (ms)
Off	Logitech MX-300 Optical USB	12.0	16.0
On	Logitech MX-300 Optical USB	35.0	38.5
Off	Dell DEL1 Optical USB	19.5	24.5
On	Dell DEL1 Optical USB	38.0	41.5

TRIGGER OUTPUT AND INPUT

It is often useful to trigger external hardware when a stimulus presentation begins. There are several ways to achieve this on typical personal computers. The parallel port can be used so that a pin goes from low to high voltage when the first frame of a stimulus is drawn. The Vision Egg has support for reading and writing to the parallel port, but because OpenGL operates in an asynchronous manner (see Section "Low-level Hardware and Software Overview: Drawing in OpenGL"), the parallel port cannot be updated at the exact instant the display begins a new frame. Instead, the parallel port can only be updated before the `swap_buffers()` command is given or after it returns. Better accuracy could be obtained by "arming" the trigger of a data acquisition device immediately before stimulus onset and triggering from the vertical sync pulse of a video cable. Ultimate verification can be done with a photodetector on a patch of screen that changes luminance at the onset of the experiment. This patch-of-screen is implemented in the `ephys_gui.pyw` application described in Section "High-level Software Overview: An Application for Electrophysiology."

Some hardware used in experiments, such as fMRI machines, has intrinsic timing requirements and thus it is advantageous for the Vision Egg to act as a slave and to begin a stimulus upon receiving a digital pulse. Because of its realtime nature, it is straightforward to achieve temporal precision equivalent to the latencies described in Section "Timing of Visual Stimuli: Measurements of Latency," although there might be slight differences in timing due to use of a parallel port for input rather than a USB mouse.

PRECISE CONTROL OF COLOR AND LUMINANCE

METHODS TO MEASURE LUMINANCE

For the measurements described below, the Vision Egg version 1.0 was running on a dual Athlon 1400 Windows 2000 system with an nVIDIA GeForce 4 Ti 4200 graphics card and an

LG Electronics Flatron 915 FT + CRT monitor at a resolution of 640 × 200 at 200 Hz. Luminance measurements were made with a silicon photometer (OptiCal with LightScan software by Cambridge Research Systems, Ltd).

RESULTS OF LUMINANCE CALIBRATION

An 8-bit per color framebuffer allows specification of 256 luminance levels for each of the three color channels (see Section "Low-level Hardware and Software Overview: Hardware"). Each red, green, and blue value is used as an index into the appropriate color lookup table, which is used by the RAMDAC to produce an analog signal. Low contrasts or other effects may be achieved, even with an 8-bit per color framebuffer, by use of a 10-bit lookup table. Non-linearities of CRT displays are well understood (for review, see Brainard et al., 2002) with the most famous non-linearity being display luminance *gamma*. The lookup tables can compensate for this gamma property such that color specified is linearly proportional to the luminance produced on the display, and the Vision Egg includes the ability to calibrate and compensate automatically for this gamma property. Finally, some computers have framebuffers with >8 bits per color. In OpenGL and the Vision Egg, colors are specified as a floating point value between 0.0 and 1.0 so the same program benefits immediately from the improved hardware.

Photometric luminance measurements of the display made with full screen color values are shown in **Figure 6**. The most well known of the non-linearities of video displays is characterized by the gamma function

$$L = kp^{\gamma}, \qquad (1)$$

with L being luminance (in cd/m^2), k being a scaling constant, p being the color value specified to OpenGL for each of the red, green, and blue components of the screen, and gamma γ. In the example shown, the uncalibrated display system had $\gamma = 2.1$. By loading the appropriate values in the color lookup tables, a linear relation between specified color value and luminance output was achieved, with $\gamma = 1.0$.

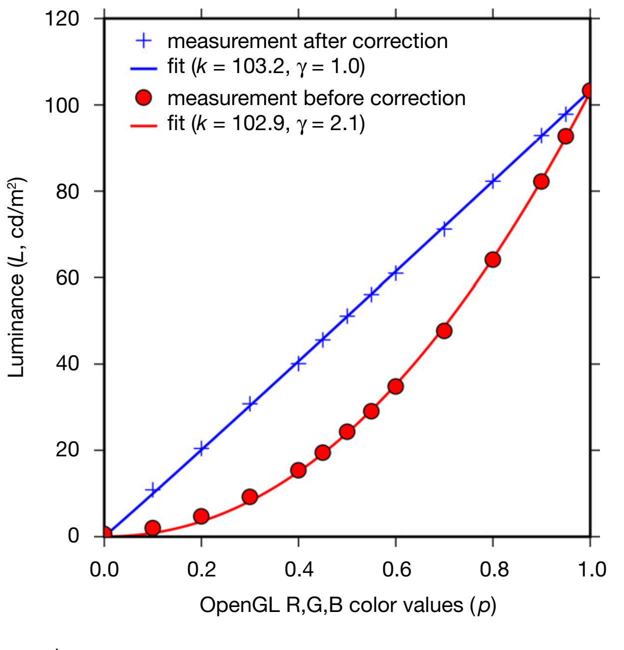


Figure 6 | **Luminance output of a CRT display is made linear with respect to commanded pixel value.** Color values specified in OpenGL units produce non-linear luminance relationship on an uncorrected display (red circles), but a corrected display has a linear relationship between specified and actual luminance (blue crosses). Lines are linear least squares fits to Eq. 1, Section "Precise Control of Color and Luminance: Results of Luminance Calibration," with coefficients given in the legend.

DISCUSSION

IMPACT OF THE VISION EGG

Although usage for open source software is notoriously difficult to estimate, the number of downloads of the Vision Egg from SourceForge.net since the first release (November 2001) totals over 15,000. Another estimate is the number of papers citing use of the Vision Egg. To date, the total listed at the website is 14. The University of Bielefeld, Germany and the University of Adelaide, Australia have used the Vision Egg in undergraduate courses (Bart Geurten and David O'Carroll, personal communication).

Other software uses or incorporates the Vision Egg. For example, in this issue, (Spacek and Swindale, 2008), describe use of the Vision Egg as part of a system for high-throughput electrophysiology. Python based extensions called *BCPy2000* to the large project *BCI2000*, a general-purpose system for brain–computer interface (BCI) research, allow customizable experiment design using the Python scripting language (Schreiner, 2008; Jeremy Hill, personal communication). SR Research developed Pylink to interface their eye tracker to Python-based software, such as the Vision Egg, and they ship a Vision Egg based example to demonstrate gaze contingent control of a moving gradient.

Finally, perhaps the greatest impact of software packages such as the Vision Egg has simply been as a proof of concept that using OpenGL and Python for creating visual stimuli is possible. Several people have told me that they looked at the Vision Egg to see how something was done and then re-implemented it themselves. Such a spread of ideas is one of the benefits of open source, although the diversity of similar but different solutions can also be a challenge, particularly for those attempting to pick a solution without investing too much in an evaluation process.

COMPARISON TO SIMILAR OPEN SOURCE SOFTWARE

PsychoPy is another Python-based open source visual stimulus system (BSD license). The author, Peirce (2007) says, "For a good programmer, Vision Egg achieves its goals very well, providing a powerful and highly optimized system for visual stimulus presentation and interactions with hardware (including the ability to run experiments remotely across a network). Straw does, however, adhere very strongly to an object-oriented model of programming which can be harder for relatively inexperienced programmers, like most scientists, to understand. For instance, the temporal control of experiments in Vision Egg is predominantly though the use of presentation loops, whereby the user sets an object to run for a given length of time, attaches stimuli to it, attaches it to a screen and then tells it to go." I believe the criticism is directed not so much toward object oriented programming (which is also employed at a fundamental level within PsychoPy) but rather Peirce's concern is with the mainloop-and-callback mechanism of flow control described in Section "Mid-level Software Overview: Controlling Program Flow." As mentioned in that section, and demonstrated in **Figure 2**, this is only optional, and the user may also maintain full control of program execution. Nevertheless, in the early development of the Vision Egg, this mainloop-and-callback style was present in all the demonstration scripts, and was intrinsic to the electrophysiology applications envisioned, like that of Section "High-level Software View: An Application for Electrophysiology." Indeed, it was a response to my own difficulties implementing psychophysics experiments with this style that I wrote demo scripts with their own flow control and began documenting the possibility.

Apart from the differences mentioned above in the style of programming, the most substantive differences today between the Vision Egg and PsychoPy are that the Vision Egg offers relatively simple perspective corrected stimuli utilizing the 3D nature of OpenGL, while PsychoPy has an automated luminance calibration utility and interfaces with Bits++ from Cambridge Research Systems, Ltd. Furthermore, the primary development platform of the Vision Egg is GNU/Linux, while it appears to be Windows for PsychoPy.

The Psychophysics Toolbox (Brainard, 1997; Pelli, 1997) has evolved greatly since the situation described in Section "Introduction: Historical Context." There is a large overlap between the possibilities offered by the PsychToolbox and the Vision Egg. Although the PsychToolbox is now officially open source (GNU GPL license), the main language of implementation is MATLAB, a proprietary application. Thus, its appeal as an open source solution is limited. Nevertheless, a core developer, Mario Kleiner, tests PsychToolbox functions with Octave, an open-source MATLAB clone, and many useful functions are implemented in C and could be used from environments other than MATLAB. Due to its heritage, most of the demonstration scripts for the PsychToolbox use pre-rendered stimuli, but it is now capable of using OpenGL and generating complex stimuli in realtime.

For another comparison between Vision Egg, PsychoPy, and the PsychToolbox, see Peirce (2007).

TOWARD A DATABASE OF VISUAL STIMULI

An online database of scripts to generate stimuli used in visual neuroscience would be useful for realizing the benefits of open-source software described in Section "Introduction: Open Source Software and Python." Other databases, such as of neuronal models (e.g., ModelDB and NeuronDB), biochemical reaction networks (e.g., SBML), and so on are proving useful in their fields. For visual neuroscience, Viperlib, an online visual perception library, might be a natural host for such a database of stimulus scripts for experiments. First, however, some serious technical issues must be solved. Although libraries like the Vision Egg and PsychoPy make it relatively easy to generate visual stimuli in a free way that is theoretically hardware independent, the issues of framerate, display luminance and position calibration, and synchronization with data acquisition and other hardware would all need to be addressed. Nevertheless, the availability of open source libraries and a number of publications based on them means that such endeavor could already be started.

CONCLUSION

The Vision Egg is a free and open-source programming library that allows scientists to produce arbitrary visual stimuli. Such stimuli can be specified in realtime without skipping frames, may involve traditional stimuli such as sinusoidal gratings, or may be more complex, 3D, and naturalistic scenes. Features such as perspective correction and realtime interpolation of image data for sub-pixel movement are part of OpenGL and thus occur in realtime at little or no extra programming or computational cost.

With the continued increase in power of conventional consumer graphics hardware, the use of such systems for vision science experiments will continue to become more common. This paper described a visual stimulus generation system that utilizes such hardware and addresses critical calibration issues in the luminance and time domains. Of course, such calibration also depends on the display device, which also has temporal, spatial, spectral, and polarization properties that need to be accounted for.

With powerful stimulus generation software and video cards now available, the greatest challenge of producing visual stimuli may now be finding an appropriate physical display device. CRTs are well understood (Bach et al., 1997; Brainard et al., 2002; Cowan, 1995) and would remain a popular stimulus presentation device, but are becoming increasingly more difficult to acquire as their production stops. LCD and DLP based devices are useful for many experiments (Packer et al., 2001). Finally, custom built LED devices may be constructed to address many issues faced with standard commercial technology (Lindemann et al., 2003; Reiser and Dickinson, 2008). Regardless of display technology, if the display device accepts standard inputs (e.g., VGA or DVI), a modular approach to stimulus generation may be used, and stimulus generation software such as the Vision Egg may be used.

CONFLICT OF INTEREST STATEMENT

The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

ACKNOWLEDGEMENTS

I thank David O'Carroll for many discussions about, much feedback on, and an environment in which to create the Vision Egg. Many others have contributed to the Vision Egg over the years with bug reports and code submissions. Sol Simpson of SR Research, in particular, helped elucidate the latency issue described in Section "Low-level Hardware and Software Overview: Drawing in OpenGL." Silicon Graphics, Inc. provided a loan of a high performance workstation. Thanks to Michael Dickinson for use of the high speed video camera. Work was partially supported by a Predoctoral fellowship from the Howard Hughes Medial Institute, who also graciously allowed a leave of absence to work for a summer in private industry, where I learned enough about realtime graphics to create the Vision Egg.

REFERENCES

Bach, M., Meigen, T., and Strasburger, H. (1997). Raster-scan cathode-ray tubes for vision research – limits of resolution in space, time and intensity, and some solutions. *Spat. Vis.* 10, 403–414.

Brainard, D. H. (1997). The psychophysics toolbox. *Spat. Vis.* 10, 433–436.

Brainard, D. H., Pelli, D. G., and Robson, T. (2002). Display characterization. In Encyclopedia of Imaging Science and Technology, J. Hornak, ed. (New York, NY, Wiley), pp. 172–188.

Cowan, W. B. (1995). Displays for vision research. In Handbook of Optics, Vol. 1: Fundamentals, Techniques, and Design, M. Bass, ed. (New York, NY, McGraw-Hill), pp. 27.21–27.44.

Fry, S. N., Müller, P., Baumann, H. J., Straw, A. D., Bichsel, M., and Robert, D. (2004). Context-dependent stimulus presentation to freely moving animals in 3d. *J. Neurosci. Methods* 135, 149–157.

Fry, S. N., Rohrseitz, N., Straw, A. D., and Dickinson, M. H. (2008). TrackFly: virtual reality for a behavioral system analysis in free-flying fruit flies. *J. Neurosci. Methods* 171, 110–117.

Kern, R., Lutterklas, M., Petereit, C., Lindemann, J. P., and Egelhaaf, M. (2001). Neuronal processing of behaviourally generated optic flow: experiments and model simulations. *Netw. Comput. Neural Syst.* 12, 351–369.

Lindemann, J. P., Kern, R., Michaelis, C., Meyer, P., van Hateren, J. H., and Egelhaaf, M. (2003). Flimax, a novel stimulus device for panoramic and highspeed presentation of behaviourally generated optic flow. *Vis. Res.* 43, 779–791.

Packer, O., Diller, L. C., Verweij, J., Lee, B. B., Pokorny, J., Williams, D. R., Dacey, D. M., and Brainard, D. H. (2001). Characterization and use of a digital light projector for vision research. *Vis. Res.* 41, 427–439.

Peirce, J. W. (2007). PsychoPy – psychophysics software in python. *J. Neurosci. Methods* 162, 8–13.

Pelli, D. G. (1997). The VideoToolbox software for visual psychophysics: transforming numbers into movies. *Spat. Vis.* 10, 437–442.

Reiser, M. B., and Dickinson, M. H. (2008). A modular display system for insect behavioral neuroscience. *J. Neurosci. Methods* 167, 127–139.

Riemersma, T. (2000). Detecting vertical retrace. *Windows Dev. J.* 11.

Schreiner, T. (2008). Development and Application of a Python Scripting Framework for bci2000. Thesis, Universität Tübingen.

Spacek, M., and Swindale, N. (2008). Python for high-throughput electrophysiology. *Front. Neuroinform.*

Straw, A. D., Rainsford, T., and O'Carroll, D. C. (2008). Contrast sensitivity of insect motion detectors to natural images. *J. Vis.* 8, 1–9.

Straw, A. D., Warrant, E. J., and O'Carroll, D. C. (2006). A bright zone in male hoverfly (*Eristalis tenax*) eyes and associated faster motion detection and increased contrast sensitivity. *J. Exp. Biol.* 209, 4339–4354.

Watson, A. B., and Pelli, D. G. (1983). QUEST: a Bayesian adaptive psychometric method. *Percept. Psychophys.* 33, 113–120.