Shady: a Software Engine for Real-Time Visual Stimulus Manipulation

N. Jeremy Hill[a,b], Scott W. J. Mooney[a,b], Edward B. Ryklin[c], Glen T. Prusky[a,b,d]

[a]*Burke Neurological Institute, White Plains, NY, USA*

[b]*Blythedale Children's Hospital, Valhalla, NY, USA*

[c]*Ryklin Software, Brooklyn, NY, USA*

[d]*Department of Physiology and Biophysics, Weill Cornell Medicine, New York, NY, USA*

Abstract

Background: Precise definition, rendering and manipulation of visual stimuli are essential in neuroscience. Rather than implementing these tasks from scratch, scientists benefit greatly from using reusable software routines from freely available toolboxes. Existing toolboxes work well when the operating system and hardware are painstakingly optimized, but may be less suited to applications that require multi-tasking (for example, closed-loop systems that involve real-time acquisition and processing of signals).

New method: We introduce a new cross-platform visual stimulus toolbox called Shady (`https://pypi.org/project/Shady`)—so called because of its heavy reliance on a *shader* program to perform parallel pixel processing on a computer's graphics processor. It was designed with an emphasis on performance robustness in multi-tasking applications under unforgiving conditions. For optimal timing performance, the CPU drawing management commands are carried out by a compiled binary engine. For configuring stimuli and controlling their changes over time, Shady provides a programmer's interface in Python, a powerful, accessible and widely-used high-level programming language.

Results: Our timing benchmark results illustrate that Shady's hybrid compiled/interpreted architecture requires less time to complete drawing operations, exhibits smaller variability in frame-to-frame timing, and hence drops fewer frames, than pure-Python solutions under matched conditions of resource contention. This performance gain comes despite an expansion of functionality (e.g. "noisy-bit" dithering as standard on all pixels and all frames, to enhance effective dynamic range) relative to previous offerings.

Conclusions: Shady simultaneously advances the functionality and performance available to scientists for rendering visual stimuli and manipulating them in real time.

1. Introduction

Neuroscience in general, and vision science in particular, place stringent demands on the ability to present visual stimuli accurately and precisely. In particular, we note the need for high timing precision, linearity, high dynamic range, and pixel-for-pixel accuracy (see section 2 below). All of these are non-trivial to achieve when one tries to program stimulus presentation software from scratch. Therefore, programmers often rely on a visual stimulus toolbox—a suite of reusable software routines that implement strategies for overcoming one or more of the aforementioned challenges, allowing the user to focus on higher-level stimulus design issues.

In this paper we present Shady[1], a visual stimulus toolbox that provides a programmer's interface in the Python programming language. It provides a two-dimensional graphics engine well-suited for rendering the most common types of stimuli in visual psychophysics: stimulus patches with carefully controlled spectral or statistical properties, "random dot" patterns (or patterns that otherwise consist of large numbers of independently-moving shapes), text, and static or animated images read from common image or movie file formats (jpeg, PNG, mpeg, avi, etc.)

Several visual stimulus toolboxes are already publicly available. Most if not all of these are listed on the frequently updated, comprehensive overview of visual psychophysics software maintained on the world-wide web by Strasburger [2]. The three general-purpose toolboxes that provide functionality most directly comparable to Shady are Psychtoolbox [3, 4], PsychoPy [5, 6, 7] and VisionEgg [8]. In common with PsychoPy and VisionEgg (but unlike Psychtoolbox which is based principally on the proprietary Matlab platform) Shady's programming interface is in Python. Python is a free, non-proprietary, versatile programming language with a large, active, worldwide community of scientists as well as non-scientists, and a rich ecosystem of scientific packages and applications.

The paper is organized as follows. In section 2 we recap some of the main problems of visual stimulus presentation that Shady and other visual stimulus toolboxes exist to solve. Then in section 3 we outline our motivation to create integrated multi-tasking neuroscience applications, and how this defined our priorities in developing Shady to be better suited for multi-tasking contexts than the established alternatives. In section 4 we outline Shady's architecture in greater detail, to show how it addresses the problems we have highlighted. In section 5 we report performance test results. In particular, we illustrate how Shady's hybrid binary/interpreted architecture makes its timing performance more robust to non-ideal conditions (specifically: resource contention and low-performing graphics hardware) relative to approaches that use pure Python. Finally, in section 6 we highlight possible avenues for further development and summarize the value of Shady to neuroscientists in its existing state, with the example of an existing project that validates Shady's ability to support accurate visual psychophysics [9].

2. Challenges in Presenting Visual Stimuli

Shady, in common with other visual stimulus toolboxes, aims to help scientific programmers overcome the many technical challenges of accurate stimulus presentation. The principal challenges are timing precision, linearization, dynamic range enhancement and pixel-for-pixel accuracy, as described in the following respective sections.

2.1. Timing precision

Neuroscientists frequently look for the ways in which stimulus changes are time-locked to other concurrently measured data, such as brain signals, eye-tracking data, and behavioural responses. This imposes the constraint that complex stimuli need to be animated smoothly, sometimes at high frame rates, with a very low tolerance for "dropped" frames—i.e. for any failure to meet the deadline to update a stimulus in time for the video hardware to display it. At a frame rate of 60 Hz, for example, such deadlines occur every 16.67 ms. They may be missed if one attempts to perform too much processing between frames—or if there is unexpected processor load. This usually results in a visible temporally-broad-band artifact or "glitch". As real-time interactive designs become more prevalent in neuroscience, timing becomes ever more critical, as computational resources must be divided between multiple processes that are all time-sensitive. Shady has two core design features that help it achieve good timing performance even under challenging conditions. First, Shady's powerful universal shader pipeline allows as much processing as possible to be performed on the computer's graphics processing unit (GPU) rather than its central processing unit (CPU). Second, to minimize the impact (and the variance in timing) of the remaining CPU operations, the main draw loop is implemented not in a high-level interpreted language (Python) but as a more-efficient binary compiled from C++, which then calls back out to Python briefly to update stimulus parameters between frames. We show some quantitative performance results in section 5 below.

2.2. Linearization

Vision science routinely demands that the pattern of luminance on screen conform to an exact mathematical specification. For example, if the experimental design calls for a sine-wave grating, the video hardware must deliver a physical luminance pattern that exactly follows a sine function. The obstacle is that most commercially available displays are not linear: there is typically a *gamma* function $L = kx^{\gamma}$ (or a function that closely approximates this shape) relating the value x in video memory to the physical luminance L actually produced. The nonlinearity γ must be measured and corrected to avoid unintended distortion products in the stimulus. Shady corrects for this, either using a generic gamma curve, or using the standard red/green/blue (sRGB) nonlinearity—a prevalent standard that uses a piecewise function, close but not identical to a gamma curve with $\gamma = 2.2$. In either case the correction is performed on the GPU as part of the standard, universal shader pipeline.

2.3. Dynamic range enhancement

Often our stimuli must have a high ratio between their energy in some spatial frequency bands and their energy in others. Our ability to achieve this is often limited by the fact that only a small number (typically 256) of discrete intensity levels are available. The rounding of each pixel intensity to the nearest integer value for the digital-to-analog converter (DAC) creates quantization artifacts. A simple example is a sine-wave grating presented at very low contrast, such that it degenerates to a square wave oscillating only between, say, DAC levels 127 and 128. This introduces higher-frequency components that may still be detectable. At even lower desired contrast levels, we lose control completely: if the baseline intensity level is close to an integer DAC value, all the pixel intensities start to be rounded to that same DAC value and the stimulus vanishes altogether; alternatively, if the baseline level falls exactly between two DAC values, all pixels will continue to be rounded to those same two DAC values and the stimulus will remain the same regardless of the requested contrast.

Hardware solutions to this problem have included specialized graphics cards that natively offer more than 8 bits (and hence more than 256 intensity levels) per colour channel, as well as analog resistor networks that combine differentially-attenuated physical voltage signals from different colour channels to compose a single monochrome input for an analog display device [10]. Modern specialized display devices such the ViewPixx monitor (VPixx Technologies) or Bits# Stimulus Processor (Cambridge Research Systems) can accomplish similar tricks in a digital implementation, sacrificing either colour or horizontal resolution to reinterpret the content of an ordinary 8-bit graphics card's video memory as a higher-dynamic-range pattern.

Software-only solutions include "bit-stealing" [11], in which the number of available luminance levels is increased by allowing the levels to deviate very slightly in chromaticity, despite the stimulus being nominally monochromatic. In areas where luminance variations as a function of distance are very gradual, this can cause large neighbouring areas to have noticeably different hues—this problem is often addressed by adding very-low-contrast pixel noise to the image before bit-stealing. Another software approach, the "noisy-bit" method of Allard and Faubert [12], simply uses low-contrast noise to "dither" between neighbouring DAC levels, independently in all three colour channels. Both of these noise-based approaches rely on the principle that our visual system locally integrates and averages variations in pixel intensity at very high spatial and temporal frequencies. By default, Shady performs noisy-bit dithering independently on every pixel, every colour channel and every frame. This is part of its standard, universal shader pipeline, and runs on the GPU without the user having to intervene, commit CPU resources, or pre-compute anything. It can be turned off to reduce the demand on GPU resources. As an alternative, Shady also provides mechanisms for performing bit-stealing, with or without additive noise, and can also reprocess pixel data on-the-fly for specialized hardware such as the ViewPixx or Bits#, increasing dynamic range at the expense of either colour or resolution. Thus, it provides multiple options for achieving higher effective dynamic range, either with or without specialized equipment.

2.4. Pixel-for-pixel accuracy

In response to a recent explosion in display hardware resolution, modern operating systems have had to develop strategies for preventing user-interface elements from rendering too small on screen. Two prevalent examples are the "DPI Scaling" feature of recent Windows versions, and the handling of "Retina" hardware by macOS. In both cases the operating system deceives software applications into believing that the screen resolution is lower than it really is, and then magnifies the rendered results on screen. Vision scientists do not benefit from this deception. When designing stimuli (for example, when assessing and minimizing effects of spatial and spatio-temporal aliasing) it is instead desirable to work with the full physical resolution of the screen in the secure knowledge that one logical pixel equates to one physical pixel (or at least an integer number of physical pixels) thereby avoiding potential artifacts from rescaling. To see through the deception and/or disable the rescaling behaviour, extra programming steps are required, with the attendant demands of careful research, implementation and debugging. When Shady opens a stimulus window with default settings, the full resolution of the screen is made available—the user should ensure that maximum resolution is selected in the operating-system preferences (at least on modern screens that have a discrete number of physical pixels) but does not need to disable the operating system's built-in rescaling tricks.

3. Design Priorities for a Modern Stimulus Toolbox

Neuroscience applications are becoming more complex and increasingly dependent on closed-loop real-time interaction. It is increasingly common for stimuli to change their properties in ways that are contingent on other ongoing operations, such as real-time electroencephalogram (EEG) or eye-tracking analysis. This multi-modal integrated approach continually demands more powerful and efficient tools. In particular, it forces each module to be a "good citizen"—i.e. to be as interoperable as possible with, and minimally intrusive on, other modules. This is particularly important when the application must be deployed practicably and portably "in the field" outside of a specialized laboratory setting—for example clinically, at the bedside. In such settings there is less scope for dedicating separate hardware for each separate task, and increased pressure to perform well on less-than-optimal hardware (which may need to be chosen for its portability at the expense of performance). Shady was designed with such factors in mind—specifically, we wanted it to:

- **Embrace Windows.** For better or worse, Windows is the platform on which we expect to find most prevalent support for specialized neuroscientific hardware (eye-trackers, EEG amplifiers, etc.) as well as novel human interface devices. This may constrain one's choice of stimulus presentation platform, especially in tightly integrated or in-the-field settings, where it may not be possible to dedicate and optimize a separate computer for stimulus presentation. Consequently, Windows is the platform on which we expect to see the greatest development of integrated multi-modal neuroscience applications, and hence the place where Shady has most to offer as a modular component of such systems. Therefore, while we ensure that all of Shady's code compiles and runs without error on macOS and Linux, we chose Windows as our primary platform for performance optimization. This is in contrast to some other stimulus toolboxes—to take two examples, VisionEgg was primarily developed on Linux with secondary support for Windows, and at the time of writing, Psychtoolbox offers some desirable features (such as real-time dithering) on macOS and Linux but not on Windows.

- **Minimize CPU demand.** Our aim was to perform as much as possible of the frame-by-frame stimulus generation and processing on the GPU. This includes functional generation of common carrier signals, spatial windowing and other forms of contrast modulation in time and space, linearization, dynamic-range enhancement, animation, and geometric stimulus transformations. This means we have greater flexibility to manipulate stimuli in real time than we would have if forced to pre-compute these operations. It also means we have greater capacity for running other concurrent tasks (such as acquisition and real-time processing of biosignals and other inputs) than we would have if the CPU had to perform these computations between frames.

- **Minimize impact on other processes.** Despite heavy reliance on the GPU, we must still perform various time-critical CPU operations on every frame. Though fast, these operations are numerous enough to provide multiple opportunities for the CPU to be interrupted by other demands. Unfortunately, on any fully-featured modern computer, background processes beyond the user's direct control will demand CPU resources on a sporadic basis, sometimes seemingly as unpredictable as the weather. As a high-level dynamic language, Python is not optimized for speed [13] and is particularly vulnerable to interruption and consequent dropping of frames. Often, designers of time-critical Python systems will try to counter this by shutting down as many other processes and services as possible, and then demanding higher-than-normal priority from the CPU at the expense of any remaining processes. In the interests of better integration into modern systems, we wished to avoid this approach. Instead we replaced Shady's core drawing loop with a binary "engine" compiled from C++. As illustrated by the performance tests in section 5 this greatly improves the robustness of Shady's timing performance, relative to any solution (including PsychoPy and VisionEgg) whose main loop is implemented in pure Python. In addition to the improvement in Shady's own performance, this also reduces the demand for higher priority and the likelihood of having to shut down other processes.

- **Maximize compatibility with other subsystems of potential neuroscience applications.** To this end, we ensure that Shady runs correctly on recent versions of Windows, macOS and Ubuntu Linux, and that it does not require any proprietary software beyond the operating system itself. We ensure that it works on both prevalent versions of the Python language (2 and 3) despite their mutual incompatibility. To avoid version conflicts among third-party dependencies, we constrained Shady's development to (a) use very few third-party Python packages, and only those that are general-purpose, well-established, widely used, and actively maintained (specifically: `numpy`, `pillow`, `matplotlib`, `ipython`, and to a lesser extent `opencv`); (b) use only the most generic core functionality of these dependencies, avoiding anything version-sensitive (we test it with 5-year-old versions of the dependencies as well as current versions), and (c) circumscribe the influence of each dependency as narrowly as possible, so that Shady's functionality degrades gracefully even in its absence.
- **Be kind to the programmers who have to use it.** A helpful strategy in optimizing interoperability is to minimize the complexity of users' code. Python, well known for its minimalism, is a good starting point to achieve this. Building on this strength, Shady is designed to minimize the amount of "boilerplate" code users have to write: it takes one short line of code to open a window and start the engine, another line to add a stimulus, and often only one line to specify how that stimulus should be animated as a function of time. Shady is also careful to avoid dominating the user's programming environment or constraining it with specialized requirements. Finally, Shady provides an interactive interface, in which users can type commands and see the immediate effects on stimuli without having to stop or restart them: this allows rapid iteration while designing stimuli or implementing and debugging experiments.

4. Architecture

Figure 1 shows an overview of Shady's architecture. The subsequent two sections explain the role of each of Shady's components, and provide a breakdown of the steps performed by the central "shader" component.

4.1. Overview of components

Shady consists of the following components (of which the principal ones are also illustrated in Figure 1):

- **Shader.** Shady is based on the Open Graphics Library (OpenGL), a highly prevalent cross-platform system for interfacing with the high-performance functionality of a graphics processor. Among other functionality, OpenGL allows one to write shader programs that are carried out on the graphics processor with a very high degree of parallelism (multiple pixel values are computed simultaneously by multiple processing cores) thereby allowing a large amount of computation in a short time. In OpenGL applications, the principal language for shader programs is GL Shading Language (GLSL) which is somewhat similar to C. At Shady's core is a single shader program, written in GLSL, that provides a flexible universal pipeline for rendering visual stimuli linearly and with enhanced dynamic range. The sequence of operations performed by the shader is described in greater detail below.

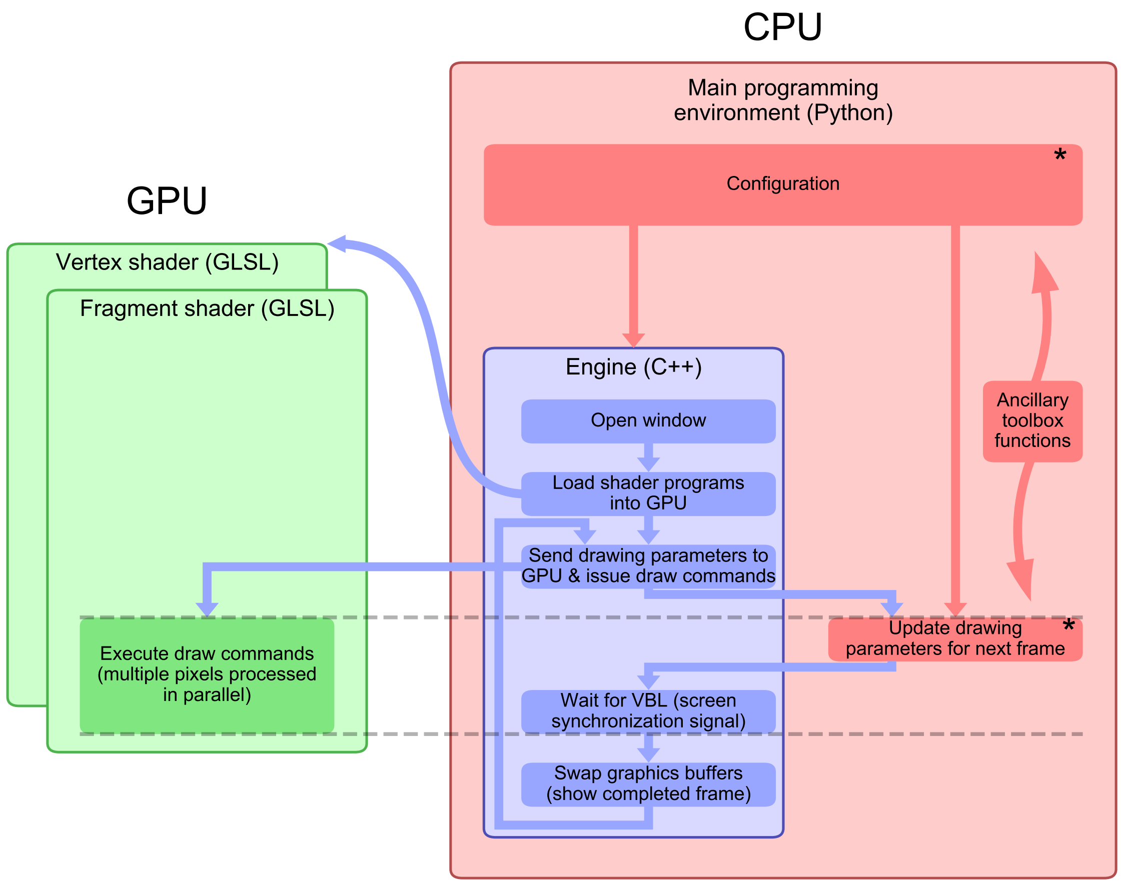
CPU
Main programming environment (Python)
Configuration
*
GPU
Vertex shader (GLSL)
Fragment shader (GLSL)
Engine (C++)
Open window
Load shader programs into GPU
Send drawing parameters to GPU & issue draw commands
Ancillary toolbox functions
Execute draw commands (multiple pixels processed in parallel)
Update drawing parameters for next frame
*
Wait for VBL (screen synchronization signal)
Swap graphics buffers (show completed frame)

Figure 1: Schematic overview of Shady's architecture, showing the role of the graphics processing unit (GPU) in running shader programs, and of the central processing unit (CPU) in running a compiled binary engine embedded within the user's Python programming environment. The dashed lines illustrate the way in which the GPU computation is performed simultaneously with two sequential CPU operations. The asterisks mark the parts of the pipeline that the user implements, in Python.

- **Programmer's interface.** Users install Shady as an add-on package to Python. Shady's Python interface makes it easy for users to configure stimulus appearance, and to write any frame-by-frame logic that dictates how stimuli change over time (the places of these two tasks in Shady's overall workflow are marked by asterisks in Figure 1). In typical Python style, it is usually possible to access powerful functionality very flexibly using very few lines of code.
- **Engine.** On every frame, the CPU must manage the transfer of stimulus parameters from main memory to graphics memory, and then issue the commands that trigger the graphics processor to run the shader and thereby draw the stimulus. The Shady "engine" is an implementation of a generic main loop that handles such housekeeping. Shady originated as, and still provides, an engine implementation in pure Python. However, this is not recommended for performance reasons, as discussed in section 3 above. Shady now also provides an "accelerated" engine—a binary compiled from C++—which it uses by default. This handles the housekeeping operations but also calls back out to Python between frames, to run the user's optional code for animating stimuli. This combination preserves flexibility for the user while minimizing variation in frame-to-frame timing (see section 5 for performance results).

- **Ancillary tools.** Shady provides a supporting ecosystem of: (a) utility functions for performing common tasks, such as conversion between pixels and degrees of visual angle, conversion between nominal and actual contrast ratios, or computation, loading and saving of look-up tables for bit-stealing; (b) high-level diagnostic tools for examining performance—for example: test patterns, timing performance checks, an interactive perceptual linearization tool, and capture of stimuli to image files, to video files, or to a real-time contrast-enhancing magnifier tool that helps in examining dithering; (c) flexible tools for controlling stimulus behavior in time: general-purpose function objects that can be manipulated arithmetically, as well as specialized functions that integrate, smooth, oscillate, self-terminate, or perform discrete state-machine transitions.
- **Interactive tutorials, demos and tests.** Shady comes packaged with a comprehensive suite of tutorial scripts that demonstrate its functionality and test its accuracy and performance. Optionally, these can be run interactively, such that the user can influence the stimuli by typing Python commands at a prompt, with results visible immediately in real time.
- **Complete source code and reference documentation.** Shady's programmer's interface—in other words, every individual class, function and method intended for direct use by Shady's user base—includes full documentation.

All of Shady's Python, GLSL and C++ code, as well as the documentation, is available to the public as a free downloadable package under the open-source GNU Public License (GPL). Our aim in using the GPL by default is to ensure that the code, and any derivative work, remains freely available and distributable in future (although we are not against potential hybrid licensing in principle). The documentation and interactive scripts can also be viewed on Shady's web-site. The main web-site address may change in future years, but we will ensure that it remains linked from the Python Package Index, via `https://pypi.org/project/Shady` .

4.2. Stimulus rendering pipeline

In Shady's central unified stimulus processing pipeline, all stimuli are thought of as being composed of a carrier presented within an envelope. The carrier may be a "texture" (i.e. a discrete array of fixed values) sourced from memory, from an on-the-fly rendering of text, or from an image file, movie file or animated GIF file. Alternatively, the carrier may be a function, written in GLSL, to be evaluated by the GPU at every pixel on every frame. In either case, the user only has to think about specifying the *ideal* values of the carrier luminance values (or RGB colour triplets) and does not have to compute explicitly how these values change according to linearization or dynamic-range enhancement tricks, or according to various envelope effects. Envelope effects (e.g. spatial windowing and contrast modulation) are also computed by the GPU in the shader. The envelope itself may be a cohesive patch, or it may be fragmented into thousands of independently moving shapes (the latter is how Shady supports random-dot stimuli).

The steps of the pipeline are as follows:

1. Compute a carrier pattern. This may be (a) a patch of solid color, (b) a pre-determined texture array, (c) a signal function that the GPU uses to generate patterns procedurally on every frame, or (d) an additive combination of these options.
2. Translate, rotate or scale the carrier pattern if requested. All carriers are treated as infinitely, cyclically repeating patterns for this purpose. In the case of procedurally-generated signals, the transformations are actually applied to the coordinate system, before the signal function is evaluated—this avoids artifacts from spatial quantization or interpolation.
3. Apply envelope effects. These may include (a) a windowing function that attenuates the edges of the stimulus, (b) another more arbitrary procedural modulation function (for example, sinusoidal contrast modulation), (c) a uniform contrast scaling factor, (d) a uniform colour tint, or (e) a multiplicative combination of these options.
4. Translate, rotate or scale the complete stimulus if desired.

5. Add noise, if desired. A two-dimensional uniform or Gaussian pixel noise pattern can be added to the stimulus at this stage. It is useful at very low amplitudes if we intend to apply a look-up table in the next step, or at higher amplitudes if we actually want a visible noise effect.
6. Apply gamma-correction and dynamic-range enhancement effects. This is done by one of the following procedures:
   a. DEFAULT OPTION: transform each channel's pixel values through the inverse of the screen non-linearity, then dither each color-channel independently between the DAC values immediately above and immediately below the target level, according to the noisy-bit algorithm;
   b. OR: transform each channel's pixel values through the inverse of the screen non-linearity, then re-represent the result as 16-bit values, dividing the more- and less-significant bytes either between colour channels or between horizontally-adjacent pixel locations in video memory—this allows specialized hardware like the ViewPixx or Bits# to render stimuli with high dynamic range by sacrificing either colour or resolution;
   c. OR: quantize pixel values according to the size of a large (say, 16-bit) pre-generated look-up table, then use the table to look up a triplet of (red,green,blue) DAC values—for monochromatic stimuli this can simultaneously accomplish linearization, bit-stealing if desired, and further quantization down to the native precision of the video hardware;

The pipeline can be customized by embedding small snippets of user-defined GLSL code into one's Python configuration commands: this allows users to extend the variety of signal functions, spatial windowing functions, and contrast modulation functions offered by the shader.

Various demo scripts, bundled with Shady, allow users to verify the accuracy of the processing pipeline for themselves. For example, by typing `python -m Shady demo precision` the user can run the "precision" script as an interactive tutorial: this script estimates and reports the accuracy and precision of step 6a in the pipeline above.

5. Timing Performance Assessment

In benchmarking Shady's performance we will focus on comparison against PsychoPy, because the latter is the most widely-used Python-based toolbox among vision researchers. In doing so, we should note that Shady's scope is very much narrower—it focuses only on visual stimuli, and only has an interface for programmers: Shady does not aim to replace PsychoPy, but might be used as an improved visual "back end" for PsychoPy and an attractive alternative to its current reliance on less-specialized, less vision-science-optimized rendering engines.

We see Shady being used in two different contexts: laboratory and field. In laboratory conditions, we assume that the stimulus presentation hardware is optimized for graphical performance; that it performs no other role besides stimulus presentation; and that we might even isolate the computer from the network, raise the CPU priority of the stimulus presentation process, and/or shut down operating-system services that are not absolutely essential to stimulus display. To be sure, part of our motivation in developing Shady was to increase the sheer amount of frame-to-frame processing we might be able to do under such ideal conditions. However, as explained in section 3, the equally important complementary motivation was to improve resilience against *less-than-ideal* conditions in the field. Under field conditions we assume that the hardware might need to perform multiple roles, some of them possibly requiring network connectivity, and that it may be purchased and configured according to multiple criteria that may require compromise on graphical performance.

5.1. Timing tests under field conditions

To provide challenging "field" conditions, we used a Surface Pro 3—a tablet computer equipped with a high-resolution screen but not a particularly high-performing GPU or CPU—running Windows 10. We closed any particularly unpredictable resource-hungry applications such as web browsers, but did not disable the wireless network connection and made no attempt to suppress background operations, for example by shutting down operating-system services. The screen mode was set to ensure maximum resolution (2160 × 1440 pixels, 32-bit color, 60 Hz refresh rate). The following further details were extracted from the output of the `dxdiag.exe` utility:

System Model:	Surface Pro 3
BIOS:	3.11.2150
Processor:	Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz (4 CPUs), ~2.3GHz
Memory:	8192MB RAM
Operating System:	Windows 10 Pro 64-bit (10.0, Build 16299)
Card name:	Intel(R) HD Graphics 5000
DAC type:	Internal
Device Type:	Full Device
Display Memory:	2160 MB
Dedicated Memory:	112 MB

At the time of these tests, the latest stable release of PsychoPy was 1.90. This version had been best-optimized for a particular 32-bit distribution of Python, version 2.7.11, with which it was bundled as an all-in-one "standalone" release. We therefore chose this Python distribution to run both PsychoPy and Shady. We ran a modified version of the timing benchmark test `timeByFramesEx.py` that is included in the PsychoPy distribution. This rendered a 300-by-300-pixel Gabor patch whose orientation and phase changed smoothly on every frame, as well as a progress bar, 1/20 of the height of the screen, whose width grew from nothing to the full width of the screen to indicate the proportion of the requested number of frames rendered so far. We made two modifications: first, we used a `Shady` utility function to disable Windows' "DPI Scaling" feature and thereby ensure that we were addressing the screen at its native physical resolution; second, we added the option of rendering and animating multiple independent copies of the Gabor simultaneously. We then recreated the same stimulus configuration and behavior using Shady, to allow head-to-head testing.

CPU fluctuations are, in part, unpredictable. Therefore, different conditions cannot be compared fairly by running them sequentially—any observed difference in performance might be the result of a sporadic increase in CPU demand, which coincided with one condition rather than another purely by chance. We therefore interleaved our tests with high temporal granularity. We launched Python and rendered a single stimulus patch for 100 frames using PsychoPy, then exited and re-launched Python to render 100 frames using Shady, then exited and re-launched for 100 frames using Shady with "acceleration" disabled (falling back on the pure-Python implementation of the engine). This inner cycle of three different implementations was then repeated using 2 stimulus patches, then with 3, then 4, then 5. This full cycle (3 implementations × 5 stimulus configurations) was repeated 25 times, so that each of the 15 conditions was tested for a total of 2500 frames (around 40 seconds total, discontiguous time).

Results are plotted in Figure 2. The number of stimulus patches increases between panels from left to right, as indicated by column titles. Different implementations are shown in different rows, as indicated by the legends: PsychoPy (using its own pure-Python drawing engine) is in the top row of panels; Shady (using its default C++ engine with callbacks to Python) is shown in the bottom row. Each panel shows frame-to-frame interval in milliseconds, plotted as a solid line as a function of frame number. Ideally, it should be a flat line at 16.67 ms, indicating 60 perfectly regular frames per second. Irregular timing shows up as "spikes" in this trace. A "dropped" frame is defined, as per the original PsychoPy diagnostic, as a frame-to-frame interval more than 1.5 times the target interval: in our case, 25 ms. The total number of spikes exceeding 25 ms is indicated by the "dropped" figure in the middle of each panel. The paler colored patch at the bottom of each panel represents the amount of time, on each frame, taken to complete the drawing commands on the CPU. Barely visible at the upper edge of that patch is a darker patch, which represents the additional time taken on each frame by the animation code (a small number of pure-Python statements, in all implementations) to update the phase and orientation of the stimuli.

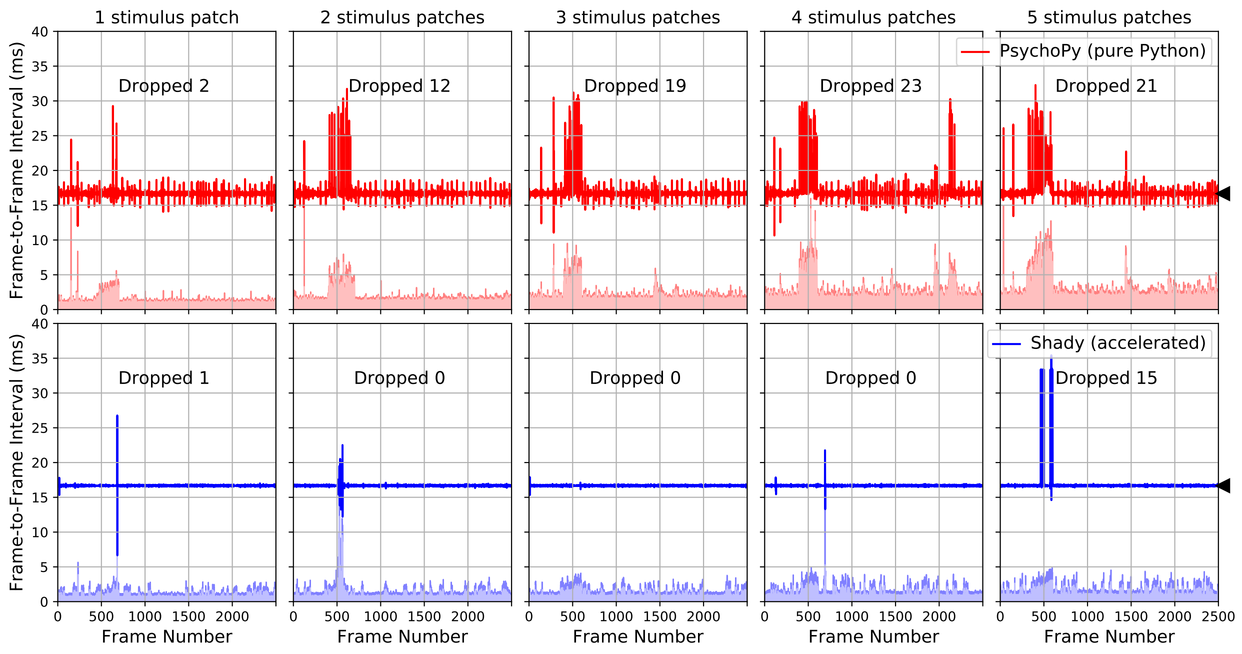

Figure 2: Comparative timing results under matched non-ideal conditions. Fifteen conditions were interleaved, 100 frames (1.67 s) at a time: 3 implementations (of which 2 are shown) × 5 stimulus configurations. Solid lines show wall time elapsed from one frame to the next, plotted on the y axis (the target—marked by the black triangles—is 16.67 ms, and anything more than 25 ms causes a visible "dropped frame" artifact). Shaded patches at the bottom of each panel indicate the time taken, on each frame, for the engine to send draw commands to the GPU and then call the routine that updated the stimulus animation between frames. Due to the interleaving, the "frame number" axis can be interpreted (very roughly, to within about ±15 s across panels) as a common time axis for all panels, spanning the total test duration of about 25 minutes. Therefore, the disturbance at around frame 500 can be attributed to the same event (presumably some kind of background operation by the operating system) in all panels.

Since we are examining the performance of a real-time system, all timing measurements use "wall time"—i.e. the time elapsed from beginning to end of an operation according to an independent clock, rather than the "processor time" allotted to the operation. When sporadic increased CPU load strikes, as it appears to have done around 1/5 of the way through the test, it causes all our CPU operations (drawing and animation) to take longer to return. The absolute size of this additional delay is not constant—it tends to be larger for operations that take longer (contain a larger number of instructions) to begin with. In turn, the extra drawing time increases the likelihood and overall frequency of dropped frames. This illustrates a double incentive for minimizing the CPU drawing overhead: not only does it leave more time free per frame on average, but it also reduces the variability, and hence the risk of dropping frames.

Note that the PsychoPy traces contain frequent spikes, each too small to cause an actual frame drop. They occur roughly 50 frames apart on average, but with high variability. On these occasions a single frame appears to be delayed by 1–2 msec, thus manifesting as a slightly higher-than-expected frame interval relative to the foregoing frame, immediately followed by a slightly lower-than-expected interval relative to the subsequent frame. We think it likely that, rather than the video frame itself experiencing the delay on the graphics card, it is really the Python code on the CPU that was delayed and that, every 50 frames or so, the delay happened to occur before the very first command of the frame—i.e. the command that looked at the clock to record the frame start time. We should note that these perturbations were largely absent on the higher-performing "laboratory" computer described in section 5.2 below.

The accelerated implementation of Shady performed more robustly under the same conditions. Overall, it took markedly less time to complete its drawing operations than PsychoPy, dropped fewer frames, and exhibited smaller variability in timing between frames (flatter lines). We should note that Shady accomplished this while also performing more processing on the GPU during each frame: in the Shady implementation, every pixel on the screen (in the backdrop as well as the stimulus patches) is automatically dithered according to the noisy-bit algorithm. The acceleration of the main drawing loop in C++ is key to this improved performance and smaller variability: the functionally-equivalent pure-Python version of the Shady engine (not shown) was very much less efficient than either accelerated Shady or PsychoPy, exhibiting numerous small spikes and also frequently failing to meet its frame deadline throughout the tests (between 566 and 1957 frames were dropped, depending on number of stimuli).

5.2. Timing tests under laboratory conditions

Our test setup for "laboratory" conditions was a system that we habitually use for visual psychophysics: a ViewPixx monitor attached as a second screen to a desktop computer running Windows 10, disconnected from the network. The screen mode was 1920 × 1200 pixels, 32-bit color, at the higher refresh rate of 120 Hz. Further system details were as follows:

System Model:	MS-7922
BIOS:	V10.6 (type: UEFI)
Processor:	Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz (8 CPUs), ~4.0GHz
Memory:	16384MB RAM
Operating System:	Windows 10 Home 64-bit (10.0, Build 17134)
Card name:	NVIDIA GeForce GTX 1060 6GB
DAC type:	Integrated RAMDAC
Device Type:	Full Device (POST)
Display Memory:	14219 MB
Dedicated Memory:	6052 MB

We repeated the tests described above, interleaving PsychoPy and the accelerated implementation of Shady. This time we varied the number of stimulus patches up to 15. Figure 3 shows the results of the five most demanding conditions (11 to 15 stimulus patches). As expected, Shady's binary engine is able to perform the draw commands on the CPU more quickly than PsychoPy's Python code. However both implementations performed perfectly: the frame-to-frame intervals were smooth and uniform, and there were no dropped frames.

In additional tests on the laboratory computer, we found that Shady could animate up to 126 separate fully-independent stimulus patches at 120 Hz before it started to drop frames. Note, however, that creating separate fully-independent stimuli is not the most efficient way to animate multiple moving elements. The more powerful way is to create a single stimulus object, specify its envelope as an array of coordinates that define separate shapes, and update the array of coordinates between frames. One of Shady's built-in demos, called "dots4", draws up to 10,000 independently-moving shapes, orbiting an origin that responds in real time to the user's mouse movement; this demo ran on both our "field" setup at 60 Hz, and on our "laboratory" setup at 120 Hz, without dropping frames.

6. Outlook and Conclusion

Shady was created as part of an ongoing project at Blythedale Children's Hospital, to support field applications—specifically, visual psychophysics and eye-tracking in children with brain injuries, performed at the bedside. This is one component of our broader vision of a "scalable neurological assessment platform" (SNAP) which consists of multiple reusable software modules. Providing full documentation and support to external users is part of our strategy to ensure that these modules remain usable and future-proof as the platform evolves. At the time of writing, the vision project is about to enter a new five-year phase, over which period we expect

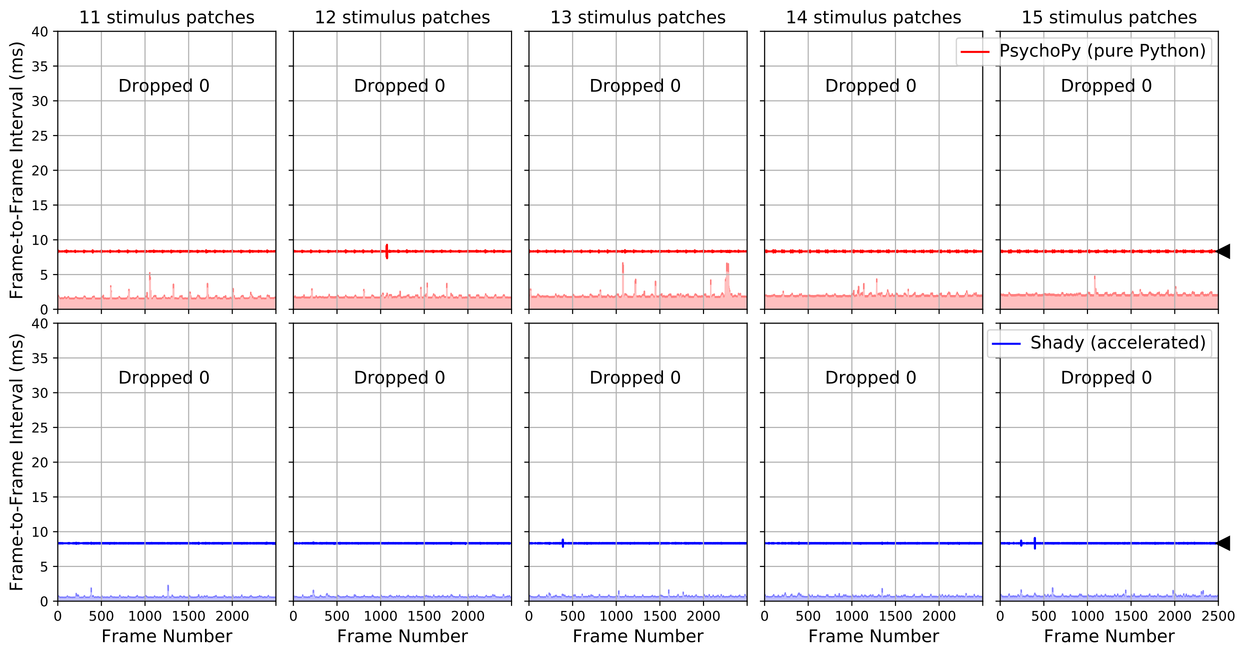

Figure 3: Comparative timing results under matched laboratory conditions. Thirty conditions were interleaved, 100 frames (0.83 s) at a time: 2 implementations × 15 stimulus configurations (of which the most demanding 5 configurations are shown). Solid lines show wall time elapsed from one frame to the next, plotted on the y axis (the target—marked by the black triangles—is 8.33 ms, and anything more than 12.5 ms causes a visible "dropped frame" artifact). Shaded patches at the bottom of each panel indicate the time taken, on each frame, for the engine to send draw commands to the GPU and then call the routine that updated the stimulus animation between frames. Due to the interleaving, the "frame number" axis can be interpreted as a common time axis for all panels, spanning the total test duration of about 25 minutes.

to be able to support users via a dedicated internet forum, currently hosted on Google Groups. Support is provided on a best-effort basis as there is, as yet, no direct funding for full-time support and maintenance.

There is room for improvement in Shady. For example, it currently focuses only on two-dimensional stimuli. Although it can be made to render three-dimensional stimuli, the user must write their own low-level OpenGL calls to accomplish this. This is similar to the way three-dimensional rendering must be done in Psychtoolbox, but it is in contrast to VisionEgg, which had high-level routines that could, for example, effortlessly wrap a grating or other texture onto a virtual cylinder. Such capabilities necessarily break the property of pixel-for-pixel accuracy discussed in section 2, but since they are required for some applications, providing them as an option is a likely goal for Shady's future development.

Another proximate goal is integration with PsychoPy. Beyond the low-level routines that render visual stimuli, PsychoPy also contains a great deal of superstructure that allows experiments to be designed, conducted, ported, archived and reproduced, even by non-programmers. It would be highly desirable for experiments implemented in Shady to be able to take advantage of this, and conversely for experiments designed in PsychoPy to take advantage of Shady's high-performance features. We have not yet begun this process, but are encouraged to note that PsychoPy's experimental design interface is in principle agnostic to the implementation, or even the programming language, for which it generates code (J. Peirce, personal communication). This is a promising early indication of feasibility.

However it may evolve in future, in its current state (version 1.8.0) Shady is ready for use by programmers who need to implement vision science experiments, or a wide range of other neuroscience applications that require visual stimuli. It is stable, and fulfills the roles it was designed to play—specifically, it enables linearized high-dynamic-range two-dimensional stimuli to be presented easily and controlled precisely, even without specialized hardware. It provides powerful, flexible tools for animating stimuli, and its timing precision rivals that of other contemporary toolboxes—in particular, it may surpass them in situations where the hardware and operating-system are less-than-optimal for high-performance graphics, or when the application requires other forms of time-critical processing to be performed concurrently. This is exemplified by the Curveball application [9], which is based on Shady. Curveball performs real-time analysis of an observer's smooth-pursuit eye movements, and uses this information for continuous adaptive adjustment of the contrast of a moving stimulus, to estimate the observer's contrast sensitivity. Curveball's' ability to manipulate stimuli smoothly in real time while also processing eye-tracking data, even on low-performance graphics hardware, illustrates Shady's success as a platform for stimulus generation and control in multi-tasking environments.

Acknowledgments

This work was supported by Blythedale Children's Hospital, and by NIH Grant R21 EY026753 awarded to authors NJH and GTP as well as Jason B. Carmel.

References

[1] Hill N. J. & Mooney S. W. J. (2018–). *Shady.* Python Package Index, `https://pypi.org/project/Shady`. Accessed on December 06, 2018.

[2] Strasburger H. (1994–2018). *Software for visual psychophysics: an overview.* Self-published web page, `http://www.hans.strasburger.de/psy_soft.html`. Accessed on December 01, 2018.

[3] Brainard D. H. (1997). *The Psychophysics Toolbox.* Spatial Vision, **10**(4): 433–436.

[4] Pelli D. G. (1997). *The VideoToolbox software for visual psychophysics: Transforming numbers into movies.* Spatial Vision, **10**(4): 437–442.

[5] Peirce J. W. (2007). *PsychoPy—psychophysics software in Python.* Journal of Neuroscience Methods, **162**(1-2): 8–13.

[6] Peirce J. W. (2008). *Generating stimuli for neuroscience using PsychoPy.* Frontiers in Neuroinformatics, **2**: 10.

[7] Peirce J, Gray J. R, Simpson S, MacAskill M, Höchenberger R, Sogo H, Kastman E. & Lindeløv J. K. (2019). *PsychoPy2: Experiments in Behavior Made Easy.* Behavior Research Methods, **51**: 195.

[8] Straw A. D. (2008). *Vision Egg: An open-source library for realtime visual stimulus generation.* Frontiers in Neuroinformatics, **2**: 4.

[9] Mooney S. W. J, Hill N. J, Tuzun M. S, Alam N. M, Carmel J. B. & Prusky G. T. (2018). *Curveball: A tool for rapid measurement of contrast sensitivity based on smooth eye movements.* Journal of Vision, **18**(12): 7.

[10] Pelli D. & Zhang L. (1991). *Accurate control of contrast on microcomputer displays.* Vision Research, **31**: 1337–50.

[11] Tyler C. (1997). *Colour bit-stealing to enhance the luminance resolution of digital displays on a single pixel basis.* Spatial Vision, **10**: 369–77.

[12] Allard R. & Faubert J. (2008). *The Noisy-Bit method for digital displays: Converting a 256 luminance resolution into a continuous resolution.* Behavior Research Methods, **40**: 735–43.

[13] Barany G. (2014). *Python interpreter performance deconstructed.* In *Proceedings of the Workshop on Dynamic Languages and Applications - Dyla'14*. ACM Press.