UNIVERSITY of CALIFORNIA
SANTA CRUZ

STIMULUS SOFTWARE REDEVELOPMENT FOR LARGE SCALE RETINAL ACTIVITY RECORDINGS

A thesis submitted in partial satisfaction of the
requirements for the degree of

BACHELOR OF SCIENCE

in

APPLIED PHYSICS

by

Anastassia Tolpygo

September 2010

The thesis of Anastassia Tolpygo is approved by:

_____	_____
Professor Alexander Sher	Professor David P. Belanger
Thesis Advisor	Senior Theses Coordinator

_____
Professor David P. Belanger
Chair, Department of Physics

Abstract

Stimulus Software Redevelopment For Large Scale Retinal Response Activity

By

Anastassia Tolpygo

A possible replacement method of the visual stimulus algorithm is explored to update experimental techniques in large-scale recording of retinal output activity. In particular, emphasis is placed on Matlab interpreted language with the Psychtoolbox platform. The benefits and limitations of the favored alternative are studied using coding manipulation and performance diagnostics supported by graphical data. Timing accuracy and hardware handshaking are essential aspects of the stimulus, in turn part of the overall recurrent cycle of the experiment. This accuracy is needed for a precise white-noise analysis method of action potentials in live retinal tissue samples. The retinal ganglion cells perform much processing before transmitting the information to the visual cortex via optic nerve. Such experimentation has helped classify retinal neurons helping further investigation of the central nervous system and its wonders.

Contents

List of Figures

List of Tables

To my mother,

Irina Tolpygo,

the most amazing person on this planet and my best friend.

Acknowledgements

I would like to greatly thank my thesis advisor Alexander Sher for providing technical and practical guidance, as well as patience through this research experience. Also, I'm grateful to all the contributing students, lab assistants, researchers and faculty that made this project accessible and understandable.

1

Introduction

1.1 Biological Background

The brain and spinal cord control and govern most functions of all living creatures. Together they are referred to as The Central Nervous System (CNS), a vast network of neuronal cells, all communicating with signals in specific patterns to execute functional or reflexive tasks. Between neurons, these patterns resemble pathways, as each signal has specific starting point and a target. This signal has been defined as an action potential (AP) and is a self-regenerating wave of electrical energy. [1] An AP is initiated in a neuronal cell body, at the axon hillock, and propagates to the terminus of the axon. [1] Each AP will travel down an axon of a neuron until it reaches its final destination at the end where the cell membrane forms a synapse with another cell.

The location at which two neurons interact is called the synaptic cleft and most commonly is an axon of the pre-synaptic cell converging at a dendrite of the post-synaptic cell. Once it reaches the end of the axon, the AP will cause the release of various neurotransmitters which will flow out into the cleft. This process can either increase or decrease the electrical potential in the membrane of the post-synaptic cell, depending on the transmitter released. Hyper- or de-polarization sensed by ion specific channels may increase or decrease the probability of an AP to spike and propagate through the post-synaptic cell; it may also cause other internal changes. The neurotransmitter process and cell anatomy is shown in Fig. 1.1.

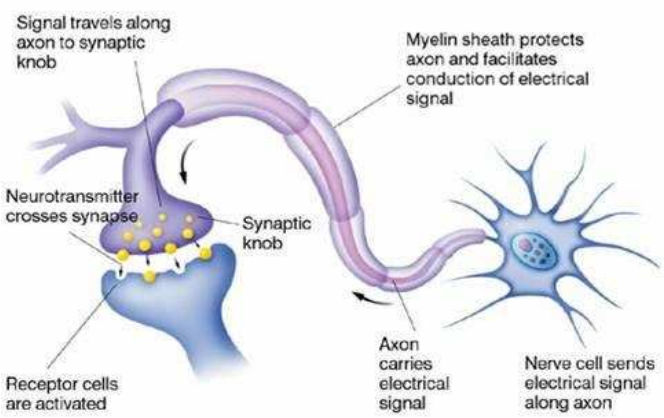

Figure 1.1 Large scale representation of a typical neuron with a close up of the synaptic cleft, adapted from http://www.ieaecell.org/epilepsy-02.html

If the membrane's voltage increases enough to reach the threshold, it will spike. Other channels will then open letting positive ions out of the cell to depolarize it until equilibrium is reached and the potential drops back to resting state. Ion concentrations controlling the opening and closing of channels clearly affect the permittivity of the cell membrane, thus this is known as 'voltage-dependent membrane permeability'. The change in voltage or a spike is shown in Fig. 1.2 below.

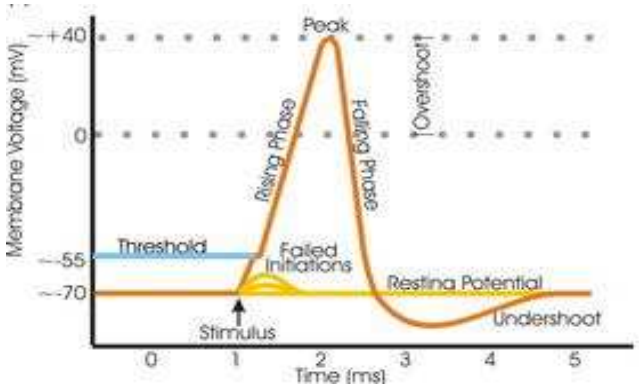

Figure 1.2 A representation of change in membrane voltage versus time with rising and falling phases and the threshold voltage emphasized, adapted from
http://en.wikipedia.org/wiki/File:Action_potential_vert.png

As can be seen in Fig.1.2, not all stimuli cause the cell to fire as the voltage must exceed the threshold. The permeation of action potentials stops at the cell or causes other internal

changes if it isn't surpassed. This operation continues through many circuits of neurons responsible for different tasks. The retina is a small gateway circuit of the CNS between the eye and visual cortexes, and has three types of neurons; Photoreceptors, Intermediate cells, and Retinal Ganglion Cells (RGCs). With interest to classify such cells by functionality, research of large scale retinal voltage activity has given insight into the overall operation of the CNS and complex brain processing.

1.2 History of Research

There have been a number of ways developed to measure the potential of a neuronal cell. Commonly used in electrophysiology, the voltage clamp method measures ion currents across a cell while holding the membrane voltage constant. Similarly, the current clamp method measures membrane voltage with constant ion current held. Another more complex method is the patch clamp method, which detects the current of just one ion channel by isolating it with a clamp. Though very useful, these methods ultimately only observe the current and voltage changes of one cell. As mentioned earlier, the CNS functions via broad circuits of neurons rather than individual cells. Hence, it is valuable to study neuronal activity on a larger scale, considering 'The Big Picture' of one or another circuit. This may be done using multiple electrodes at once, positioned outside cells. The placement of the electrodes is important for a sufficient signal to noise ratio; the closer the electrode is to a cell, the larger and more detailed the signal detected from it. Extracellular multi-electrode array recording has become the ideal way to achieve simultaneous recording of many cells. A large number of electrodes increases the

efficiency in a reading, sometimes with spacing comparable to the cells themselves; electrodes spacing as close as sixty microns. This method proves to be more useful in studying networks of neurons and has thus far helped comprehensively describe five and identify one of a couple dozen morphologically distinguished RGCs in the primate retina. **"**The action potential generated by the cell produces local ionic currents in the saline solution the retina rests on top, which in turn produce a local potential at the site of the electrode [2]." The hardware design for gathering data with multiple electrodes was inspired by high-energy particle detection; commonalities between silicon strip detectors and electrodes include dense spacing and ability to record low magnitude signals.

1.3 Hardware

As mentioned above, "the technology employed is based on silicon microstrip detector techniques and expertise developed for experiments in high-energy physics [3]." The application specific integrated circuit, or ASIC, is the common denominator relating the two fields of research. Specifically in signal acquisition, where for particle detection a silicon strip detector is mounted onto a PCB, an array of 512 electrodes is mounted on a similar board for the Neuroproject, called the Neuroboard [3]. In both cases, the signals to be gathered are nominally of small magnitude and dense spacing. These characteristics call for immediate amplification and filtering, which is performed directly with the preliminary electronics rather than with other detached hardware through which smaller waveforms may get lost in noise. The Neuroboard is the main piece of equipment that handles these functions. In Fig.1.3 the array of rectangular dimension 32 x 16 electrodes

resides in the middle, surrounded by eight Platchips and eight Neurochips. “Each Platchip has 64 channels of 150-pF capacitors for ac-coupling the electrode signal to the corresponding Neurochip channel [3].” Every Platchip also has a current generator electroplate the probes with platinum to reduce the impedence between the electrode and tissue and circuitry to provide a current pulse for neuron stimulation. [3] Subsequently, the Neurochip has 64 channels that differentially pre and post amplify as well as filter the signal, finally driving it out through a 64:1 multiplexer. The band pass range is nominally within 80-2000 Hertz.

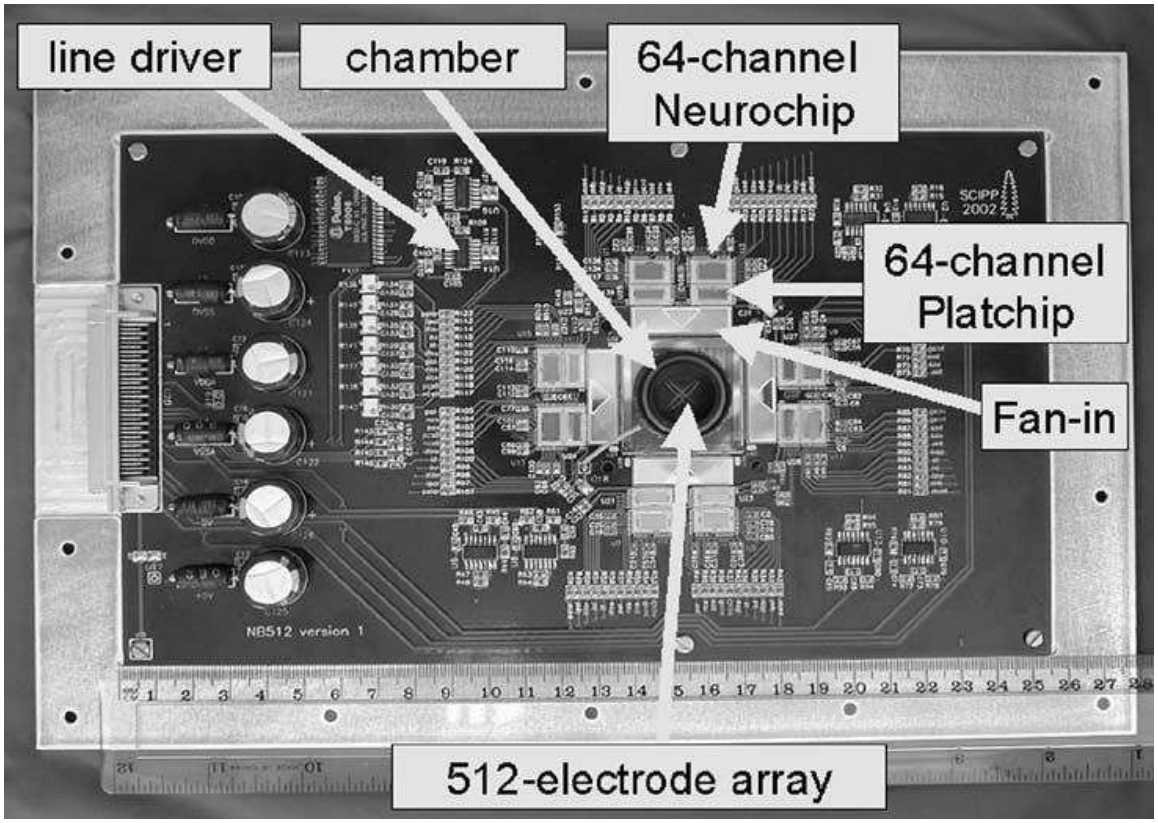

Figure 1.3 The Neuroboard at approximately 25.5cm in length, adapted from Litke et al[3].

Surrounding these main components are resistors that divide voltage to supply controlled current to each Neurochip and capacitors that filter noise. These smaller components are all surface mounted and were precisely soldered for six upgraded boards over the course

of a summer. The experimental technique is based on the work of Meister et al. and is schematically represented by Fig.1.4 [4].

1.4 Stimulating the Retina

The retina works as a processing relay between the visual world and the brain. Photoreceptive cells convert kinetic energy of the incoming photons into chemical energy to activate a train of reactions to the main neurons in the retina, the RGCs. After significant processing, RGCs' action potentials carry visual information to various targets in the brain via optic nerve. In order to identify and measure the functional specificity of the retina on a broad scale, a micro-electrode array (Fig.1.3) gathers the electrical output of RGCs induced by a visual stimulus or movie projected onto the retina (Fig.1.4).

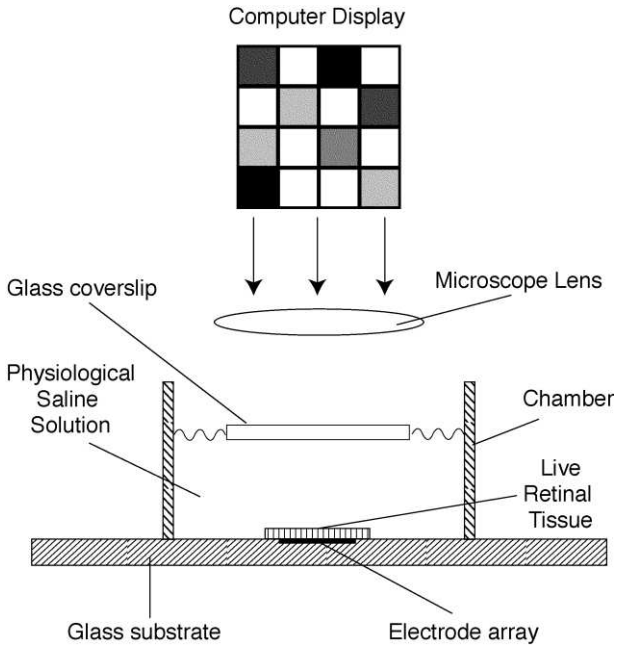

Figure 1.4 Experimental setup, adapted from Litke et al [3]

The stimulus in this case, is presented on a monitor using a computer running modifiable code specific to the desired stimulus.

Furthermore, the retina is a high functioning, complexly responsive unit which makes interpretation difficult. Primarily, the direct cause, or stimulus that triggers a certain behavior or spike train should be well known for accurate analysis. Hence, the stimulus presented has to be timely and very precise; which becomes a non-trivial software and hardware correlation task. Such an implication can complicate data analysis if the presentation time of a frame and its triggered RGC response timing do not correspond. Various tests and diagnostics were performed in order to track the precision of the algorithm by investigating the average time between each frame shown. These were done using a standard white noise stimulus which is discussed in detail in Section 1.5.

White noise stimulus is one stimulus technique in exploring the response properties of spiking visual system neurons [5]. This somewhat simple yet distinct stimulus offers the possibility to mildly stimulate many RGCs using variation in intensity and wavelength, thus generating a wide range of spatial, temporal and chromatic response characteristics. “It provides a complete and easily interpretable model of responses even for neurons that display a common form of response nonlinearity that precludes classical linear systems analysis [5].”

1.5 STA Analysis and White Noise Stimulus

As previously discussed, white noise is most commonly implemented as stimulus for studying retinal activity, causing a wide variety of responses. “As it is stochastic, highly interleaved, can span a wide range of visual inputs, is robust to fluctuations in response characteristics, and does not affect cell adaptation to strong or prolonged stimuli, this type of visual stimulant is well suited to simultaneous measurement of multiple neurons and has many advantages to other types [5].” Applied along with a refined mathematical method called Spike-Triggered Average, or (STA) that estimates the stimulus-cell response relationship, this method is useful for investigating circuits of neurons. For the retina, complex and nonlinear filtering of incoming information is the motivation behind the STA analysis. Each RGC responds to and processes particular parts of any given input. This functioning is similar to the encoding done by analog-to-digital converters together with band-pass filters to isolate specific parts of incoming information. Unlike conductors that simply pass or store information, RGCs filter and manipulate it so as to translate into a language a specific neural circuit can interpret.

A typical white noise frame is shown in Fig.1.5, a small portion of that frame that an RGC is sensitive to is called the receptive field. The response properties of that RGC are estimated using basic linear algebra for STA analysis. Reactions are due to certain parameters of encoding in a retinal cell. Such as location and size of each cell’s special receptive field, range of detectable stimulus time frequency and the preferred variation in intensity and saturation. These are known as spatial, temporal and chromatic response properties of the retina respectively [5]. One RGC may have sensitivity in all or just one

of these properties, ignoring everything except very distinct bits of information. This response distinction is achieved using white noise stimulus, because its intent is to project all the combinations of spectrum, time frequency and sizes possible of any receptive field. Determined by a normal distribution, these combinations vary through frames during prolonged running experiments.

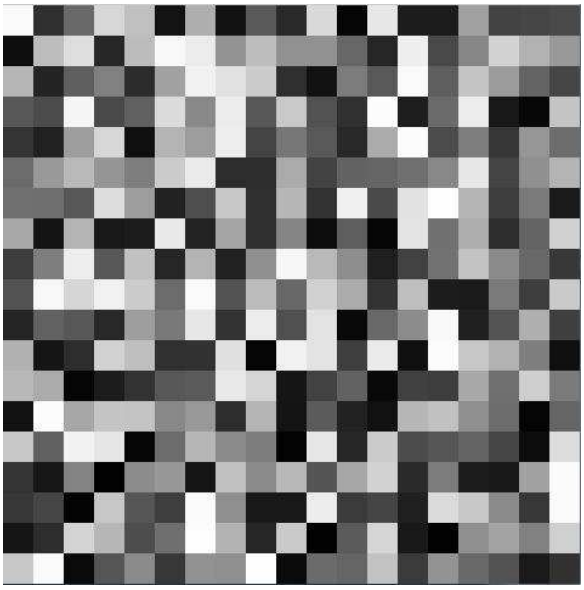

Figure 1.5 An example frame of white noise stimulus at a large Scale factor.

Consequently, STA analysis or reverse correlation will look at one cell's electrical activity and average the frames prefacing action potentials in order to calculate the parameters that cause them. STA stimulus is defined as the average stimulus preceding a spike in the cell, i.e. "the sum of the stimuli before each response, divided by total number of spikes [5]."

$$a = \left(\sum_{t=1}^{T} s_t f_t\right) / \left(\sum_{t=1}^{T} f_t\right) \quad (1)$$

The vector s_t is defined as the stimulus at a given time t and f_t is the spike count in the time immediately following this stimulus [5]. There is dependency in the response f on the stimulus s, defined as the expected response R(s):

$$R(s) = < f|s > = \sum_f fP(f|s) \quad . \qquad (2)$$

Here, P is the probability of stimulus s firing and producing a response f. It is also assumed the history of responses has no effect on this expression, which means, that the spikes are generated by a Poisson process with a mean parameter equal to R [5]. As an example, consider that the total response current in a neuron depends linearly in the stimulus, but spike probability has nonlinear dependence on current due to the physiology of the ion channels that control spike generation [5]. R(s), known as the static nonlinear model of response will incorporate such a discrepancy.

The numerator and denominator of equation (1) can be divided by the total recording time T. Taking T to infinity; the denominator becomes the 'average firing rate' and assumes a nonzero limit <f>. The same limit is assumed for the numerator. Hence, for large values of T, using equation (1) and (2) and simplifying with the use of the probability identity P(s & f) = P(s)P(f|s), the STA can be rephrased.

$$a = (1/<f>)\sum_s sP(s)R(s) \qquad (3)$$

"Thus, the STA approaches a sum of stimulus vectors, each weighted by its probability and the average response it induces, normalized by the average firing rate [5]."

As is evident, the STA approximation is heavily dependent on a timing accuracy of the stimulus frames, proportional to the firing rate, or frequency of spikes in a response train. Therefore, the nominally trivial monitor refresh rate and projected frame rate

become of utmost importance in this experiment. Since, the stimulus is generated by a computer program; its accuracy should be accounted for in that algorithm.

2

The Need to Revamp Preexisting Software and Hardware

As previously mentioned, a good way to interpret the functionality of a neural circuit is simultaneously, from many neurons. The "Neuroproject" does just that by using high energy physics instrumentation together with neurobiological experimentation and statistical data analysis to probe the mysteries of the CNS and the retina. Overall, the process is a flow of information through hardware and software paths. Initially, visual stimulus is presented to the retina. The data aquisition computer initiates the start of the stimulus program via a TTL pulse to the stimulus computer. The retina will 'see' the movie and react accordingly to the various parameters of a white noise stimulus. It rests on top of a 512 microelectrode array, where RGC spikes are captured by the electrodes. As discussed in Section 1.3, the array is mounted onto the Neuroboard. After being filtered and amplified within the circuitry, the signals are finally digitized back in the data acquisition PC which processes the incoming information with LabView. The existing hardware to software synchronization is a TTL pulse. It is sent from the stimulus to the

DAQ computer every 100 frames displayed in order to monitor the accuracy of the stimulus timing, mentioned in Section 2.1.

In general, the stimulus is generated with an algorithm implementing a pseudo-random number generator that will return a number between 0-256. The image created is an NxM matrix of 'stixels' each representing one random number and is visually described by Fig.1.5. A stixel is a predetermined square area of pixels all taking on the same random value, short for the 'stimulation pixel'. Each random value represents a grayscale color, 0 being black and 255, as white. Though irrelevant to this thesis, a stixel can also have an RGB value with three random numbers for each stimulation pixel, also on the scale of 0 – 255, 255 representing the most saturated color value. A matrix is calculated for every frame projected on a screen at some update rate which should be in sync with the monitor refresh rate. This 'on the fly' calculation takes minimal memory space and has become the preferred generating method due to the wide range of possible stimuli with variable parameters for any experimental duration and scaling.

2.1 Methods

Currently, the stimulus uses a MacOS-9 running on MacG4 with the stimulus produced using an older computer language Lisp together with C and is shown on a Sony Trinitron MultiscanE100 monitor. This system was originally implemented for many reasons, the prominent feature being the ability to turn system interrupts off, which prohibits the computer from running other functions during a stimulus presentation. The need for newer and faster hardware is now eminent and unfortunately, none of the

replacement options have such control as it is a feature of MacOS-09 only. Within the existing code, there is also some monitoring of the stimulus accuracy checking for 'missed frames', defined in Section 2.3, which is communicated back to the data acquisition computer via TTL pulse. Other experimenters in the field, such as Michael Stryker at UCSF, have begun using Matlab7.x along with a software package called Psychtoolbox-3 or PTB-3. "Matlab is a high-level interpreted language with extensive support for numerical calculations. The Psychophysics Toolbox interfaces between Matlab and the computer hardware. The Psychtoolbox's core routines provide access to the display frame buffer and color lookup table, allow synchronization with the vertical retrace, support millisecond timing, and facilitate the collection of observer responses [6]." PTB-3 also offers many demo programs with various examples of stimuli. The most compatible one offered is the 'Fast Noise Demo' (FND), which renders white noise. Formation of the image is also done with pseudo-random number generator, with each number representing the grayscale value of a stixel, named noisel in the original code [Appendix A]. There are two functions most responsible for this illustration; Screen('Flip') with built in time monitoring and Screen('MakeTexture'). The former flips the screen display from the current frame to the next while maintaining an update rate that is in sync with the vertical retrace of the monitor. The later converts the NxM matrix of random numbers into a stixel image. With these functions two adjustable variables arise; Scale determines the square area of pixels in a stixel and RectSize, the stixel dimension of the matrix. Though this new system is user friendly and manageable with upgraded hardware, problems with timing accuracy still take place. An exploration of these issues follows.

3

Problems and Diagnostics of Psychtoolbox

Unlike the older operating system, PCs intended to replace the stimulus control do not have system interrupts. Without this crucial feature, a computer has freedom to decide whether an execution is vital or not and bypass it in order to run updates, scans, etc. If per say, stimulus is running and the operating system needs to execute one of these internal maintenance functions, the stimulus may lose priority and a frame will not get flipped in time and will stay displayed past the monitor refresh. This is referred to as missing of a frame and can also occur when the computer simply does not have enough time to calculate the NxM random value matrix and misses the opportunity to change the image. Whereas, instead of displaying one frame for 11.7 milliseconds with the corresponding 85Hz monitor refresh interval, the frame will remain on the screen for 23.4 millisecond or longer, this is investigated in detail in Section 3.2. As discussed earlier, this is an enormous problem for the STA analysis, which relies heavily on the precise time correlation between stimulus and response. To clarify, it is of utmost importance to know exactly which frame precedes a certain response from an RGC in the form of action potentials. If for instance, a spike is assumed to be a result of some frame, timing of that frame must correlate exactly prior to that spike. Otherwise, it is impossible to understand what may or may not cause an RGC to spike. If it is impossible to produce perfect stimuli, it is helpful to at least monitor the existence and occurrence of missing frames to adjust analysis accordingly.

3.1 Investigating Update Rate

As an initial experiment, the Fast Noise Demo was executed multiple times with varied Scale and RectSize in order to observe the accuracy limitations of the update rate. Timing data was acquired using the built-in GetSecs timestamp, which keeps track of the system time at the start and end of the stimulus and 'count' which is the number of frames displayed. This was done for two computers with different hardware and software specifications. Figure 3.1 is that of a an older Windows XP Pro on a Dell Precision Workstation 650 with an Intel® Xeon™ 2.66GHz, double CPU processor and display on Dell M991 Monitor using a Radeon 700 series video card. This specific setup was studied with an 800 x 600 resolution setting and a monitor refresh interval of 85Hz. As seen in Fig.3.1, the x-axis is the stixel dimension of the noise patch and the y-axis is the returned update rate in Hertz. It is most evident that a plateau of the update rate exists, where neither scale nor rectSize affect the overall timing performance. Unfortunately, with more complex or larger stimuli, this accuracy breaks down and the update rate looses precision. In fact, it lessens to approximately half of the monitor refresh rate, analogous to the computer missing one display sweep and waiting for the next one. This will result in frames displayed every other monitor refresh. The cause of this is rather intuitive as the computer has more random numbers to calculate with larger noise matrices yet in the same time period. In Fig.3.1, this happens at about 200 by 200 stixels. Peculiarly, at the smallest scale setting of one, the timing breakdown arises at a larger square stixel specification. Though the patch dimension is the same with the equal amount of numbers

to generate, further investigation could not be conducted due to software malfunction. Figure 3.2 depicts a similar pattern gathered from a Windows XP Pro Dell Precision Workstation690 with Intel® Xeon® 2.66GHZ, quadruple CPU processor using NVIDIA Quadro NVX235 video card. The display ran on a Sony Triniton MultiscanE100 monitor with the highest possible refresh interval of 60 Hertz, though a less direct comparison. In this case, the falloff occurs later to agree with the assumption that the newer video card and computer are faster.

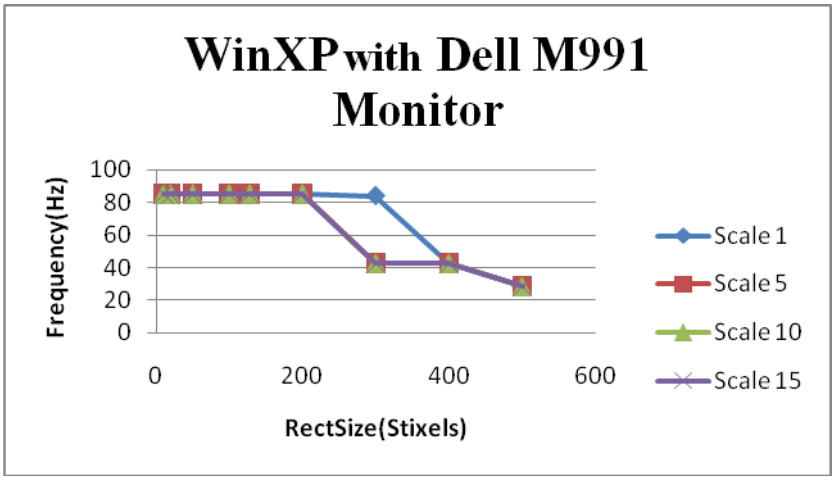

Figure 3.1 Update rate frequency versus patch size, varying in scale with a monitor refresh rate of 85Hz.

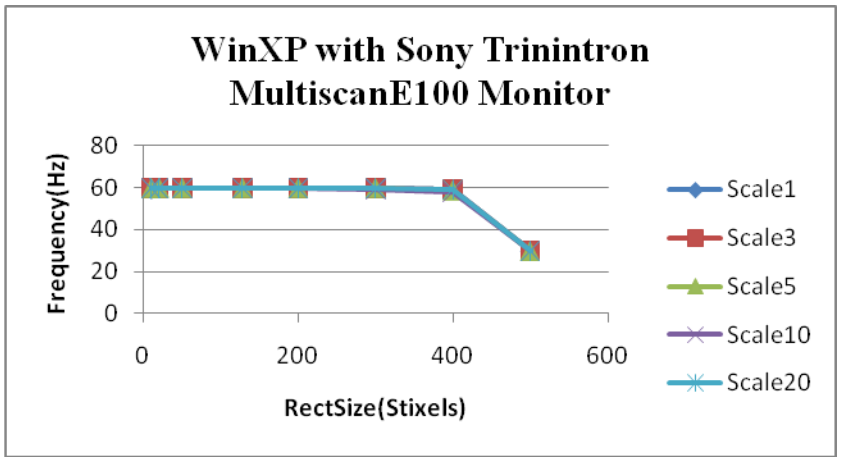

Figure 3.2 Plateau at set monitor refresh rate of 60Hz. Consistent break down of update rate to half of monitor refresh rate at approximately 400 square stixels.

The data above observes computing limitations the replacement technique may have. If the stimulus frames cannot flip in sync with the vertical retrace of the monitor, frames will be missed.

3.2 Timestamps and Missed Frames

Given the certain range of parameters that guarantee an accurate update rate found in Fig.3.1 and Fig.3.2, it is important to verify timely frame display within those restrictions. Consequently, the discovered falloff region was studied to see how many frames exactly were missed as well as the possibility to track them. Using the returned update rate and some additional code [Appendix B], missed frames were counted and recorded for the newer computer at a scale of three, a monitor refresh rate of 60Hz and a resolution of 800x600 pixels. According to previous measurements, scaling of 400 and above was expected to miss many more frames, so the duration of the experiment was reduced. The results are shown in Table 3.1.

Frame count/duration	**RectSize**	**Number of Missed Frames**
100,000/30mins	200	2
100,000/30mins	300	0
100,000/30mins	400	1
18,000/5mins	500	18000

Table 3.1 Missed frames at various dimensions.

The most important outcome here is the plateau region performing with almost ideal accuracy resulting in little to no missed frames. Though, as soon as the noise patch

increased to 500 by 500 every single frame was delayed, since the frame count was 18000. That difference between 160,000 and 250,000 random numbers to be generated forces the computer to spend more time calculating the NxM matrix which turns out to be slower than the monitor refresh rate. This is a computational restriction to be considered when choosing stimulus parameters since the computer lags behind the desired display change frequency.

Next, the original Fast Noise Demo was modified using preexisting timestamps. Along with Screen('Flip'), the algorithm can monitor the timing between frames for various stimuli specifications using VBLTimestamps. As previously stated, the major functionality of Screen('Flip') is to switch an existing front display with an anticipated back display in sync with the vertical retrace of the monitor while tracking system time when the actual flip has happened as a VBLTimestamp return argument. FlipTimestamp is another one taken at the end of 'Flip' execution. As can be seen in the modified code [Appendix B], these time stamps were implemented for every count, at the start of each displayed frame. The inverse of the update frequency defines the ideal duration that a frame ought to be shown,

$$T = \frac{1}{f}. \tag{4}$$

Hence, as mentioned earlier, at an 85Hz monitor refresh rate a frame should be displayed for approximately 11.7 milliseconds. With this definition, if a frame is missed, it can only be flipped at the next monitor refresh. So, its display time becomes a multiple of the expected time,

$$\Delta T = CT. \tag{5}$$

In this definition C is some whole number, depending on how delayed the flip is. As with most software timing, there will be some error to this exact calculation, where the timestamp will be taken slightly prior to or after the display flip, which accounts for the distribution in Fig.3.3. Using the built in Matlab histogram function and defining the interval between timestamps as the bins, timing within the plateau region was studied. Figure 3.3 shows a histogram of four built in PTB timestamps representing a stable stimulus with no missing frames but with subtle jitter around the expected time, defined by equation (4). Due to its accuracy, VBL timestamp was most useful for this experiment. Figure 3.4 shows a histogram of a small number of missed frames with the display interval up to double the expected amount.

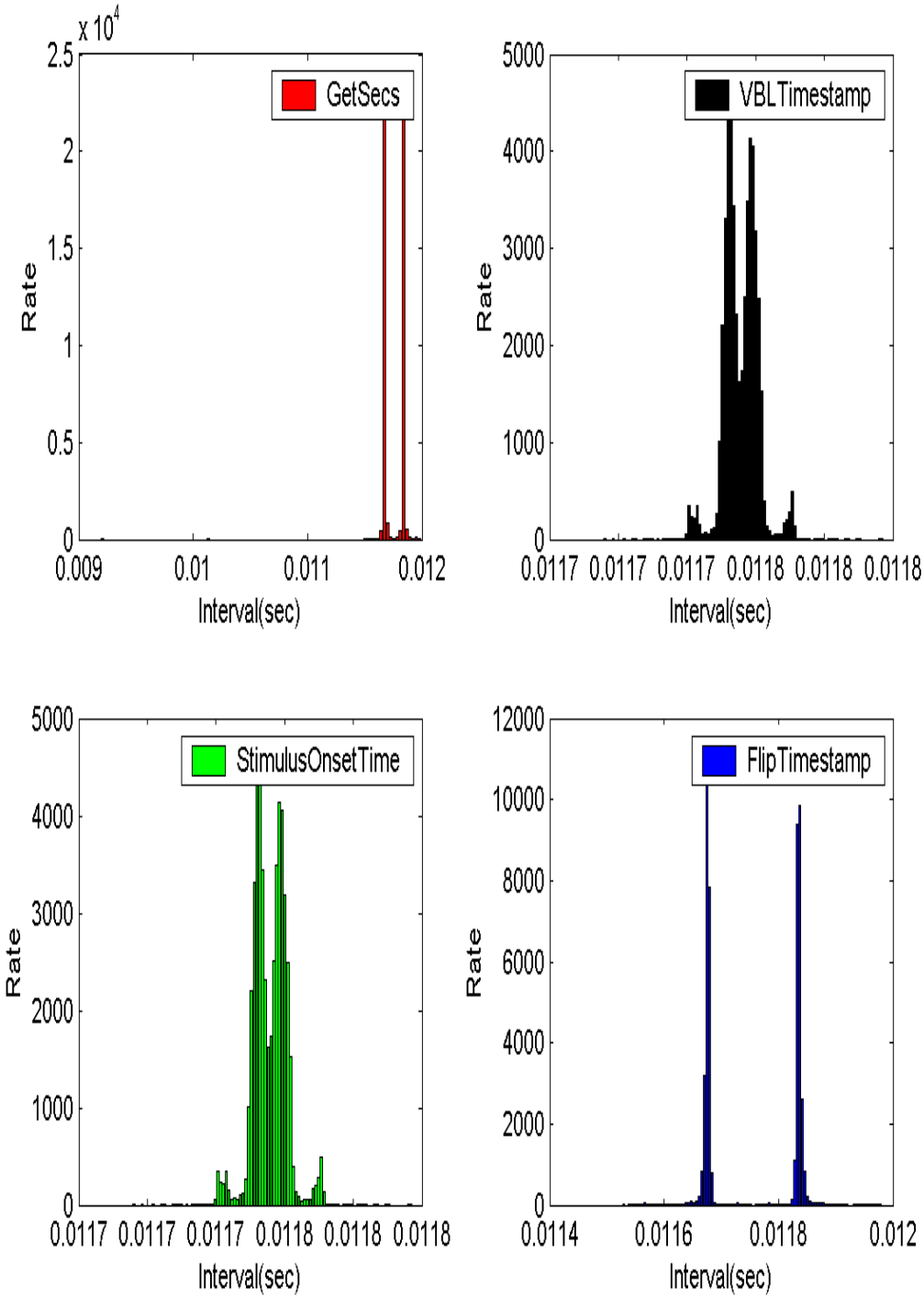

Figure 3.3 Histograms from different time stamps; StimulusOnset and VBL are identical. Note Flip and GetSecs are precise but lack distribution accuracy and were therefore less useful. The x-axis in all the histograms is in seconds but with only three significant figures max display option. The y-axis represents the number of frames displayed at the intervals.

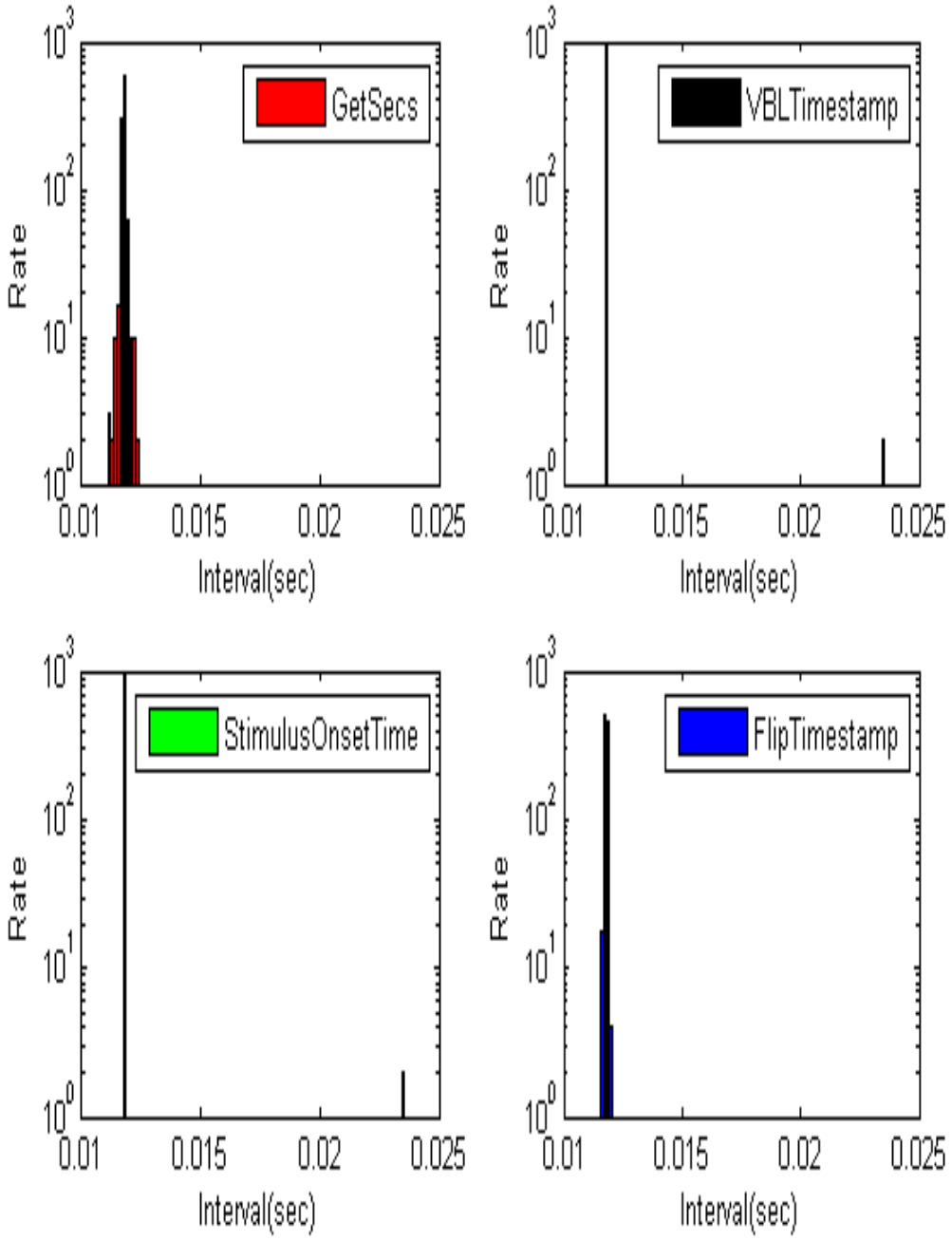

Figure 3.4 At double the expected interval, corresponding to half the monitor refresh rate, a few counts are seen. For the VBL time stamp there are a small number of counts with the display duration of 23.4 microseconds, which is more obvious when taking the y-axis into logarithmic scale.

This diagnostic was also taken at a dimension of 32x16 stixels, corresponding to the size and shape of the electrode array, and can be seen in Figure 3.5.

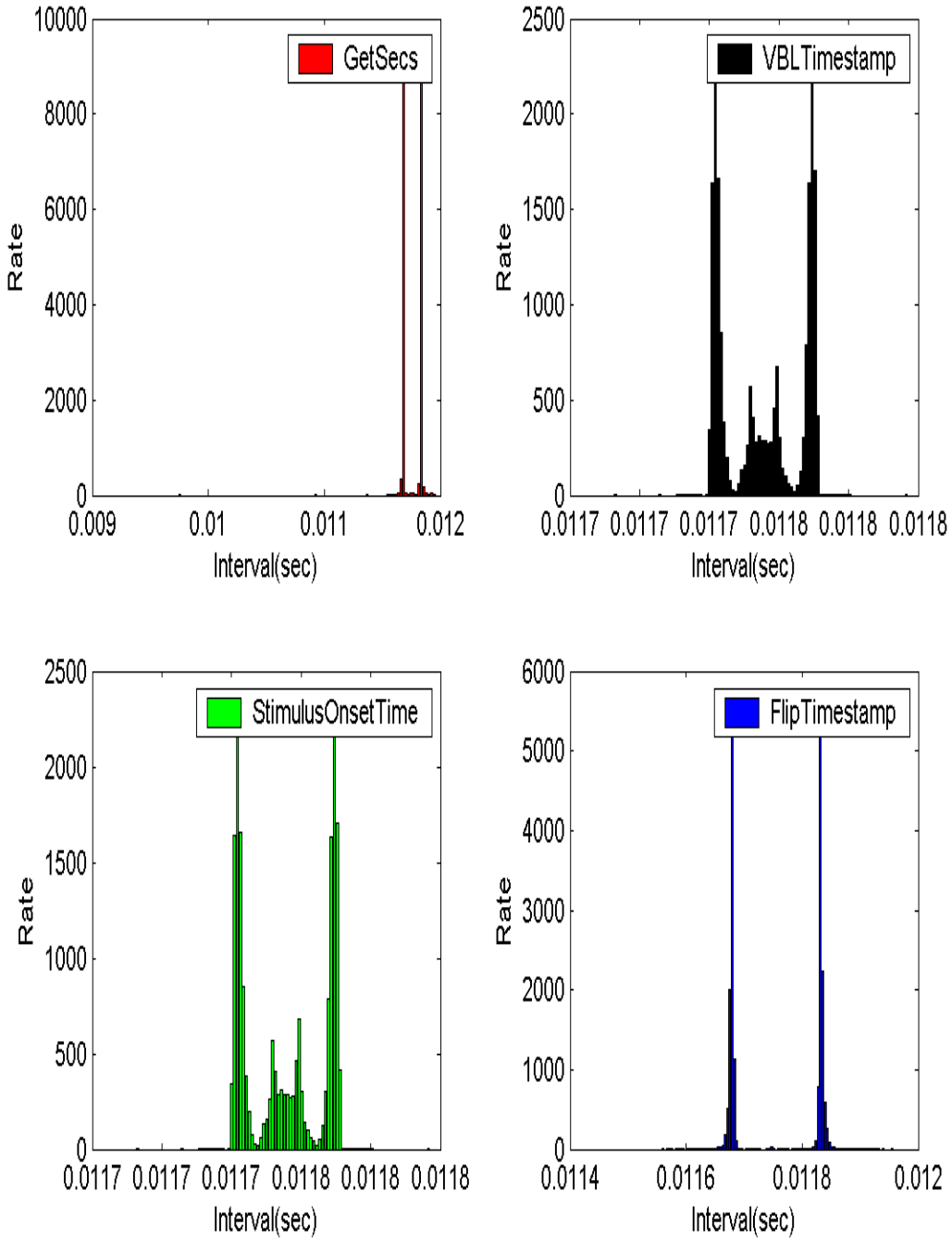

Figure 3.5 As expected, within the plateau region of zero to approximately 400 scaling, VBL timestamps are still accurate and in sync with the monitor refresh rate.

Though rather conservative, the results above give some insight into the software performance limitations in a new stimulus system intended for the Neuroproject upgrade. If missed frames are unavoidable, the ability to at least observe their occurrence is the next best thing. With the incidence of a missed frame known, the analysis can be modified accordingly.

4

Discussion

In the last twenty-some-odd years, retinal output activity investigation has been able to classify a number of retinal ganglion cells using simultaneous multi-electrode recording. The 512 electrode technique first used in 2003/04 and has since made the classification of another RGC possible. Numerous stimulation methods have been utilized, concluding Spike Triggered Average as the most enlightening for studying action potentials of retinal ganglion cells. Santa Cruz Institute of Particle Physics with the Biology department at University of California, Santa Cruz is the site of such collaboration under the guidance of Alan Litke, Alexander Sher and others. Currently, the project is undergoing a make-over of hardware and software systems. The hardware to software relay coding and electronics progress was possible with the help of Daniele Fusi and Vitalyi Fedeyev. With the outdated Lysp/C stimulus code providing many useful control functions, a robust yet accessible replacement system is needed. Matlab with the Psychtoolbox platform poses a good option, but as any technology has accuracy limitations; specifically in display exactness and time monitoring. FastNoiseDemo of white noise, used for STA, was mainly implemented in examining these problems in search of possible solutions and general adequacy of Matlab as alternate stimulus software for the Neuroproject. There is still much work to be done, such as a brand new code, rather than a provided demo. Also, the exact occurrence of a missed framed should be obtainable within the algorithm to drive precise analysis of action potential trains

recorded from live retinal samples. In order for Psychtoolbox to become the primary stimulus source, issues with timing and synchronization need to be addressed. The above results that I have gathered, have helped understand the issues in renewing the stimulus technique. It became evident that calculation limitations occur at a certain complexity of stimulus, which may be fixable with a faster processor. Also, with the scaling breakdown shown in Fig.3.1, video card limitations came to light. Perhaps, a faster computer and a more advanced video card together will supply timely and accurate stimulus without missing frames. Although if that does happen, it should be well addressed in analysis in a way to compensate or eliminate the error. Overall, with well developed software and hardware alterations, more refined classification of other RGCs and neurons of the CNS will be possible. Lastly, the newer Neuroboards, mentioned in Section 1.3, I contributed in surface mounting are currently being used in experiments.

5

Works Cited

[1] D. Purves et al eds, Neuroscience, 4th Ed. Sunderland: Sinauer Associates, Inc, pp37-119, 2008.

[2] W. Dabrowski et al, "Development of front-end ASICs for imaging neuronal activity in live tissue," Nuclear Instruments and Methods in Physics Research, A 541 pp. 405–411, 2005.

[3] A. Litke et al, "What Does the Eye Tell the Brain?: Development of a System for the Large-Scale Recording of Retinal Output Activity," IEEE Transactions On Nuclear Science, vol. 51, no. 4, August 2004.

[4] M. Meister, J. Pine, and D. A. Baylor, "Multi-neuronal signals from the retina: acquisition and analysis," J. Neurosci. Methods, vol. 51, pp. 95–106, 1994.

[5] E J Chichilnisky, "A simple white noise analysis of neuronal light responses," Computation in. Neural Systems, 12, pp. 199-213, 2001.

[6] Brainard, D. H. "The Psychophysics Toolbox," Spatial Vision 10:433-436, 1997. Can be found at: http://psychtoolbox.org/wikka.php?wakka=PsychtoolboxOverview

6

Appendix

A Fast Noise Demo Original Code [6]

```
function FastNoiseDemo(numRects, rectSize, scale, syncToVBL,
dontclear)
% FastNoiseDemo([numRects=1][, rectSize=128][, scale=1][,
syncToVBL=1][, dontclear=0])
%
% Demonstrates how to generate and draw noise patches on-the-fly in
a fast way. Can be
% used to benchmark your system by varying the load. If you like
this demo
% then also have a look at FastMaskedNoiseDemo that shows how to
% efficiently draw a masked stimulus by use of alpha-blending.
%
% numRects = Number of random patches to generate and draw per
frame.
%
% rectSize = Size of the generated random noise image: rectSize by
rectSize
%            pixels. This is also the size of the Psychtoolbox noise
%            texture.
%
% scale = Scalefactor to apply to texture during drawing: E.g. if
you'd set
% scale = 2, then each noise pixel would be replicated to draw an
image
% that is twice the width and height of the input noise image. In
this
% demo, a nearest neighbour filter is applied, i.e., pixels are just
% replicated, not bilinearly filtered -- Important to preserve
statistical
% independence of the random pixel values!
%
% syncToVBL = 1=Synchronize bufferswaps to retrace. 0=Swap
immediately when
% drawing is finished. Value zero is useful for benchmarking the
whole
% system, because your measured framerate will not be limited by the
% monitor refresh rate -- Gives you a feeling of how much headroom
is left
% in your loop.
%
% dontclear = If set to 1 then the backbuffer is not automatically
cleared
```

```
% to background color after a flip. Can save up to 1 millisecond on
old
% graphics hardware.
%
% Example results on a Intel Pentium-4 3.2 Ghz machine with a NVidia
% GeForce 7800 GTX graphics card, running under M$-Windows XP SP3:
%
% Two patches, 256 by 256 noise pixels each, scaled by any factor
between 1
% and 5 yields a redraw rate of 100 Hz.
%
% One patch, 256 by 256 noise pixels, scaled by any factor between 1
% and 5 yields a redraw rate of 196 Hz.
%
% Two patches, 128 by 128 noise pixels each, scaled by any factor
between 1
% and 5 yields a redraw rate of 360 - 380 Hz.
%
% One patch, 128 by 128 noise pixels, scaled by any factor between 1
% and 5 yields a redraw rate of 670 Hz.

% Abort script if it isn't executed on Psychtoolbox-3:
AssertOpenGL;

% Assign default values for all unspecified input parameters:

if nargin < 1 || isempty(numRects)
    numRects = 1; % Draw one noise patch by default.
end

if nargin < 2 || isempty(rectSize)
    rectSize = 128; % Default patch size is 128 by 128 noisels.
end

if nargin < 3 || isempty(scale)
    scale = 1; % Don't up- or downscale patch by default.
end

if nargin < 4 || isempty(syncToVBL)
    syncToVBL = 1; % Synchronize to vertical retrace by default.
end

if syncToVBL > 0
    asyncflag = 0;
else
    asyncflag = 2;
end

if nargin < 5 || isempty(dontclear)
    dontclear = 0; % Clear backbuffer to background color by default
after each bufferswap.
end

if dontclear > 0
```

```
    % A value of 2 will prevent any change to the backbuffer after a
    % bufferswap. In that case it is your responsibility to take
care of
    % that, but you'll might save up to 1 millisecond.
    dontclear = 2;
end

try
    % Find screen with maximal index:
    screenid = max(Screen('Screens'));

    % Open fullscreen onscreen window on that screen. Background
color is
    % gray, double buffering is enabled. Return a 'win'dowhandle and
a
    % rectangle 'winRect' which defines the size of the window:
    [win, winRect] = Screen('OpenWindow', screenid, 128);

    % Compute destination rectangle locations for the random noise
patches:

    % 'objRect' is a rectangle of the size 'rectSize' by 'rectSize'
pixels of
    % our Matlab noise image matrix:
    objRect = SetRect(0,0, rectSize, rectSize);

    % ArrangeRects creates 'numRects' copies of 'objRect', all
nicely
    % arranged / distributed in our window of size 'winRect':
    dstRect = ArrangeRects(numRects, objRect, winRect);

    % Now we rescale all rects: They are scaled in size by a factor
'scale':
    for i=1:numRects
        % Compute center position [xc,yc] of the i'th rectangle:
        [xc, yc] = RectCenter(dstRect(i,:));
        % Create a new rectange, centered at the same position, but
'scale'
        % times the size of our pixel noise matrix 'objRect':
        dstRect(i,:)=CenterRectOnPoint(objRect * scale, xc, yc);
    end

    % Init framecounter to zero and take initial timestamp:
    count = 0;
    tstart = GetSecs;

    % Run noise image drawing loop for 1000 frames:
    while count < 1000
        % Generate and draw 'numRects' noise images:
        for i=1:numRects
            % Compute noiseimg noise image matrix with Matlab:
            % Normally distributed noise with mean 128 and stddev.
50, each
            % pixel computed independently:
```

```
            noiseimg=(50*randn(rectSize, rectSize) + 128);

            % Convert it to a texture 'tex':
            tex=Screen('MakeTexture', win, noiseimg);

            % Draw the texture into the screen location defined by
the
            % destination rectangle 'dstRect(i,:)'. If dstRect is
bigger
            % than our noise image 'noiseimg', PTB will
automatically
            % up-scale the noise image. We set the 'filterMode' flag
for
            % drawing of the noise image to zero: This way the
bilinear
            % filter gets disabled and replaced by standard nearest
            % neighbour filtering. This is important to preserve the
            % statistical independence of the noise pixels in the
noise
            % texture! The default bilinear filtering would
introduce local
            % correlations when scaling is applied:
            Screen('DrawTexture', win, tex, [], dstRect(i,:), [],
0);

            % After drawing, we can discard the noise texture.
            Screen('Close', tex);
        end

        % Done with drawing the noise patches to the backbuffer:
Initiate
        % buffer-swap. If 'asyncflag' is zero, buffer swap will be
        % synchronized to vertical retrace. If 'asyncflag' is 2,
bufferswap
        % will happen immediately -- Only useful for benchmarking!
        Screen('Flip', win, 0, dontclear, asyncflag);

        % Increase our frame counter:
        count = count + 1;
    end

    % We're done: Output average framerate:
    telapsed = GetSecs - tstart
    updaterate = count / telapsed

    % Done. Close Screen, release all ressouces:
    Screen('CloseAll');
catch
    % Our usual error handler: Close screen and then...
    Screen('CloseAll');
    % ... rethrow the error.
    psychrethrow(psychlasterror);
end
```

B Modified Fast Noise Demo Code

```
function FastNoiseDemo(numRects, rectSize, scale, syncToVBL,
dontclear)
% FastNoiseDemo([numRects=1][, rectSize=128][, scale=1][,
syncToVBL=1][, dontclear=0])
% Demonstrates how to generate and draw noise patches on-the-fly in
a fast way. Can be
% used to benchmark your system by varying the load. If you like
this demo
% then also have a look at FastMaskedNoiseDemo that shows how to
% efficiently draw a masked stimulus by use of alpha-blending.
% numRects = Number of random patches to generate and draw per
frame.
%
% rectSize = Size of the generated random noise image: rectSize by
rectSize
%            pixels. This is also the size of the Psychtoolbox noise
%            texture.
%
% scale = Scalefactor to apply to texture during drawing: E.g. if
you'd set
% scale = 2, then each noise pixel would be replicated to draw an
image
% that is twice the width and height of the input noise image. In
this
% demo, a nearest neighbour filter is applied, i.e., pixels are just
% replicated, not bilinearly filtered -- Important to preserve
statistical
% independence of the random pixel values!
%
% syncToVBL = 1=Synchronize bufferswaps to retrace. 0=Swap
immediately when
% drawing is finished. Value zero is useful for benchmarking the
whole
% system, because your measured framerate will not be limited by the
% monitor refresh rate -- Gives you a feeling of how much headroom
is left
% in your loop.
%
% dontclear = If set to 1 then the backbuffer is not automatically
cleared
% to background color after a flip. Can save up to 1 millisecond on
old
% graphics hardware.
%
% Example results on a Intel Pentium-4 3.2 Ghz machine with a NVidia
% GeForce 7800 GTX graphics card, running under M$-Windows XP SP3
%
% Two patches, 256 by 256 noise pixels each, scaled by any factor
between 1
% and 5 yields a redraw rate of 100 Hz.
```

```
%
% One patch, 256 by 256 noise pixels, scaled by any factor between 1
% and 5 yields a redraw rate of 196 Hz.
%
% Two patches, 128 by 128 noise pixels each, scaled by any factor
between 1
% and 5 yields a redraw rate of 360 - 380 Hz.
%
% One patch, 128 by 128 noise pixels, scaled by any factor between 1
% and 5 yields a redraw rate of 670 Hz.

% Abort script if it isn't executed on Psychtoolbox-3:
AssertOpenGL;

% Assign default values for all unspecified input parameters:

if nargin < 1 || isempty(numRects)
    numRects = 1; % Draw one noise patch by default.
end

if nargin < 2 || isempty(rectSize)
    rectSize = 250; % Default patch size is 128 by 128 noisels.
end

if nargin < 3 || isempty(scale)
    scale = 5; % Don't up- or downscale patch by default.
end

if nargin < 4 || isempty(syncToVBL)
    syncToVBL = 1; % Synchronize to vertical retrace by default.
end

if syncToVBL > 0
    asyncflag = 0;
else
    asyncflag = 2;
end

if nargin < 5 || isempty(dontclear)
    dontclear = 0; % Clear backbuffer to background color by default
after each bufferswap.
end

if dontclear > 0
    % A value of 2 will prevent any change to the backbuffer after a
    % bufferswap. In that case it is your responsibility to take
care of
    % that, but you'll might save up to 1 millisecond.
    dontclear = 2;
end
    Priority(1);
    nframes = 1000;

try
```

```
    % Find screen with maximal index:
    screenid = max(Screen('Screens'));

    % Open fullscreen onscreen window on that screen. Background
color is
    % gray, double buffering is enabled. Return a 'win'dowhandle and
a
    % rectangle 'winRect' which defines the size of the window:
    [win, winRect] = Screen('OpenWindow', screenid, 128);

    % Compute destination rectangle locations for the random noise
patches:

    % 'objRect' is a rectangle of the size 'rectSize' by 'rectSize'
pixels of
    % our Matlab noise image matrix:
    objRect = SetRect(0,0,rectSize, rectSize);
    % ArrangeRects creates 'numRects' copies of 'objRect', all
nicely
    % arranged / distributed in our window of size 'winRect':
    dstRect = ArrangeRects(numRects, objRect, winRect);

    % Now we rescale all rects: They are scaled in size by a factor
'scale':
    for i=1:numRects
        % Compute center position [xc,yc] of the i'th rectangle:
        [xc, yc] = RectCenter(dstRect(i,:));
        % Create a new rectange, centered at the same position, but
'scale'
        % times the size of our pixel noise matrix 'objRect':
        dstRect(i,:)=CenterRectOnPoint(objRect * scale, xc, yc);
    end

    % Init framecounter to zero and take initial timestamp, define
other timing variables:

    count = 0;
    tstart = GetSecs;
    previousGetSecs = tstart;
    currentGetSecs = tstart;
    [VBLTimestamp, StimulusOnsetTime, FlipTimestamp] =
Screen('Flip',   win, 0, dontclear, asyncflag);
    previousVBL = VBLTimestamp;
    currentVBL = VBLTimestamp;
    previousOnset = StimulusOnsetTime;
    currentOnset = StimulusOnsetTime;
    previousFlip = FlipTimestamp;
    currentFlip = FlipTimestamp;
    x = 1:nframes;
    y = 1:nframes;
    z = 1:nframes;
    n = 1:nframes;

    % Run noise image drawing loop for 100 frames:
```

```
    while count < nframes
        % Generate and draw 'numRects' noise images:
        for i=1:numRects
            % Compute noiseimg noise image matrix with Matlab:
            % Normally distributed noise with mean 128 and stddev.
50,         % each
            % pixel computed independently:
            noiseimg=(50*randn(rectSize, rectSize)+ 128);

            % Convert it to a texture 'tex':
            tex=Screen('MakeTexture', win, noiseimg);

            % Draw the texture into the screen location defined by
the
            % destination rectangle 'dstRect(i,:)'. If dstRect is
bigger
            % than our noise image 'noiseimg', PTB will
automatically
            % up-scale the noise image. We set the 'filterMode' flag
for
            % drawing of the noise image to zero: This way the
bilinear
            % filter gets disabled and replaced by standard nearest
            % neighbour filtering. This is important to preserve the
            % statistical independence of the noise pixels in the
noise
            % texture! The default bilinear filtering would
introduce         % local
            % correlations when scaling is applied:
            Screen('DrawTexture', win, tex, [], dstRect(i,:), [],
0);

          % After drawing, we can discard the noise texture.
            Screen('Close', tex);

        end

        % Done with drawing the noise patches to the backbuffer:
Initiate
        % buffer-swap. If 'asyncflag' is zero, buffer swap will be
        % synchronized to vertical retrace. If 'asyncflag' is 2,
      bufferswap
        % will happen immediately -- Only useful for benchmarking!

        %Systemtiming
        currentGetSecs = GetSecs;
        count;
        tmid = currentGetSecs - previousGetSecs;
        previousGetSecs = currentGetSecs;
        x(count+1) = tmid;

        %DefineTimestamps:
        [VBLTimestamp, StimulusOnsetTime, FlipTimestamp ] =
      Screen('Flip', win, 0, dontclear, asyncflag);
```

```
    VBLTimestamp = VBLTimestamp;
    StimulusOnsetTime = StimulusOnsetTime;
    FlipTimestamp = FlipTimestamp;

    %VBL Timing:
    currentVBL = VBLTimestamp;
    VBLmid = currentVBL - previousVBL;
    previousVBL = currentVBL;
    y(count+1) = VBLmid;

    %Onset Timing:
    currentOnset = StimulusOnsetTime;
    Onsetmid = currentOnset - previousOnset;
    previousOnset = currentOnset;
    n(count+1) = Onsetmid;

    %Flip Timing:
    currentFlip = FlipTimestamp;
    Flipmid = currentFlip - previousFlip;
    previousFlip = currentFlip;
    z(count+1) = Flipmid;

    % Increase our frame counter:
    count = count + 1;

end

%Lost frames loop

format long;
outlier = 0;
for j = y;
    if  j > 0.0118;
        j
        outlier = outlier + 1;
    end
end
outlier
outlierfreq = outlier/nframes

%Display results
%x-y
%n-y
%z-y
%x, n, z
%y

% calculate statistics:
mux = mean(x);
xstd = std(x);
muy = mean(y);
ystd = std(y);
mun = mean(n);
nstd = std(n);
```

```
muz = mean(z);
zstd = std(z);
muy
%mun
%muz
%mux
%xstd
%ystd
%nstd
%zdts
%m=min(x)
%M=max(x)

%Display Histograms:

%Timestamp Histogram:
format long;
figure(2)
[n1,xout]= hist(x,100);
subplot(2,2,1);  bar(xout, n1, 'r')
legend('GetSecs')
xlabel('Interval(sec)')
ylabel('Rate')
[n2,xout] = hist(y, 100);
subplot(2,2,2); bar(xout, n2, 'k')
legend('VBLTimestamp')
xlabel('Interval(sec)')
ylabel('Rate')
[n3, xout] = hist(n, 100);
subplot(2,2,3); bar(xout, n3, 'g')
legend('StimulusOnsetTime')
xlabel('Interval(sec)')
ylabel('Rate')
[n4,xout] = hist(z, 100);
subplot(2,2,4); bar(xout, n4, 'b')
legend('FlipTimestamp')
xlabel('Interval(sec)')
ylabel('Rate')

%Timestamp difference Histogram:
format long;
figure(3)
[n1,xout] = hist(x-y, 100);
subplot(2,2,1); bar(xout, n1,'r')
legend('GetSecs - VBL')
xlabel('Interval(sec)')
ylabel('Rate')
[n2, xout] = hist(z-x, 100);
subplot(2,2,2) ; bar(xout, n2, 'g')
legend('Flip - GetSecs')
%[n3, xout] = hist(n-y, 100);
%subplot(2,2,3); bar(xout, n3, 'g')
%legend('Onset - VBL')
xlabel('Interval(sec)')
```

```
    ylabel('Rate')
    [n4,xout] = hist(z-y, 100);
    subplot(2,2,4); bar(xout, n4, 'b')
    legend('Flip - VBL')
    xlabel('Interval(sec)')
    ylabel('Rate')


    % We're done: Output average framerate:
    telapsed = GetSecs - tstart
    updaterate = count / telapsed

    % Done. Close Screen, release all ressouces:
    Screen('CloseAll');
catch
    % Our usual error handler: Close screen and then...
    Screen('CloseAll');
    % ... rethrow the error.
    psychrethrow(psychlasterror);
end
```