USE: An integrative suite for temporally-precise psychophysical experiments in virtual environments

Watson, Marcus. R.,[1] Voloh, Benjamin,[2] Thomas, Christopher,[2] Womelsdorf, Thilo.[1,2]

1: Department of Biology, Centre for Vision Research, York University, Toronto, ON, M6J 1P3, Canada

2: Department of Psychology, Vanderbilt University, Nashville, TN 37240

Corresponding Author: Marcus R Watson, watsonmr@yorku.ca

Author's note:

This work was supported by grant MOP 102482 from the Canadian Institutes of Health Research (TW) and by the Natural Sciences and Engineering Research Council of Canada Brain in Action CREATE-IRTG program (MRW, TW). The funders had no role in study design, data collection and analysis, the decision to publish, or the preparation of this manuscript. The authors would like to thank Steven Chen for help in developing and testing the software described here.

1 Abstract

The Unified Suite for Experiments (*USE*) is a complete hardware and software suite for the design and control of behavioral neuroscience experiments. The software, developed using the Unity 3D video game engine, supports both dynamic activities in immersive 3D environments, and static 2D tasks commonly used in more traditional visual experiments. The hardware, based around an Arduino Mega2560 board, enables the sub-millisecond integration and synchronization of multiple data streams. A hierarchical, modular architecture makes USE a flexible framework to realize experiments, with (i) variable nested task structures used in complex cognitive and reinforcement learning tasks, (ii) flexible use of 3D versus 2D scenes and objects, (iii) precise online control of touchscreen-, button-, joystick- and eye movement behavior, and (iv) a complete offline reconstruction of experiments for post-processing and sub-millisecond temporal alignment of data streams. In summary, USE provides an integrated, open source framework for a wide variety of behavioral neuroscience experiments in human and nonhuman subjects.

2 Introduction

2.1 Static and dynamic tasks

Participants in most psychology or neuroscience experiments are presented with highly impoverished stimuli and tightly-restricted response options. This approach, which for the sake of brevity we refer to as *static*, maximizes experimental control, constrains possible interpretations of results, allows for the relatively easy creation of experimental tasks, and in short, is a critical part of the reductionist approach that has led to many of the exceptional successes of the cognitive sciences. However, researchers are increasingly concerned with understanding behaviour and neural processing in tasks and contexts that are more complex and naturalistic, an approach we refer as *dynamic*. This paper presents an integrated suite of software and hardware designed to aid in the development, control and analysis of dynamic cognitive neuroscience experiments, the *Unified Suite for Experiments* (*USE*).

Given the success of static tasks and their dominance in the cognitive sciences, why should researchers bother with dynamic tasks? We provide three answers to this question.

First, there is the common concern about ecological validity and the generalizability of results, namely that results from simple static tasks do not easily generalize and may be misleading when taken to provide insight into real-world behaviour and neural activity (Kingstone, Smilek, & Eastwood, 2008; Schmuckler, 2001; Chaytor & Schmitter-Edgecombe, 2003). For example, there are substantial quantitative and qualitative differences in biases for gaze to be directed towards others' eyes that depend on whether participants are viewing static pictures, watching movies, or are involved in genuine interactions with other humans (Risko, Laidlaw, Freeth, Foulsham, & Kingstone, 2012). By facilitating the generation of both static and dynamic tasks using the same underlying task structure, USE enables the direct investigation of their differences.

Second, more naturalistic stimuli and flexible possibilities of control can be more immersive and dramatically more motivating for human and nonhuman primate subjects (Witmer & Singer, 1998; Youngblut, 2007; Slater & Wilbur, 1997; Bouchard, Guitard, Bernier, & Robillard, 2011; Bennett, Perkins, Tenpas, Reinebach, & Pierre, 2016). As testament to this, over the decades where video games developed more realistic visuals and more flexible and responsive controls, they developed from an obscure hobby to a multi-billion dollar industry with more consumers and larger revenues than film-making (The NDP Group, 2009; Shanley, 2017).

Finally, dynamic tasks enable the collection and exploration of a wide array of precise and multi-modal data about complex behaviors, which are necessary to generate hypotheses for understanding real-world behaviors (Kingstone et al., 2008). Despite more than two centuries

of formal psychological research and millennia of informal speculation, there is no well-established body of fine-grained data about human action in most tasks. Thus, dynamic tasks can support both hypothesis-testing and observation.

2.2 Developing dynamic tasks with USE

Dynamic tasks may be appealing, but they are more challenging to develop, control, and analyze than static tasks. USE was developed to make these challenges more manageable. It should be considered as an alternative to experimental creation suites such as Psychtoolbox (Brainard, 1997; Kleiner, Brainard, & Pelli, 2007; Pelli, 1997), PsychoPy (Peirce, 2007; Peirce, 2008), or MonkeyLogic (Asaad & Eskandar, 2008a; Asaad & Eskandar, 2008b; Asaad, Santhanam, McClellan, & Freedman, 2013), with the specific focus of creating, controlling and analyzing tasks that have the complexity, dynamism, and visual fidelity typical of video games. In addition to complementing existing dynamic experimental frameworks (Doucet, Gulli, & Martinez-Trujillo, 2016; Jangraw, Johri, Gribetz, & Sajda, 2014), USE provides additional solutions to common challenges of experimental control (see 5.2).

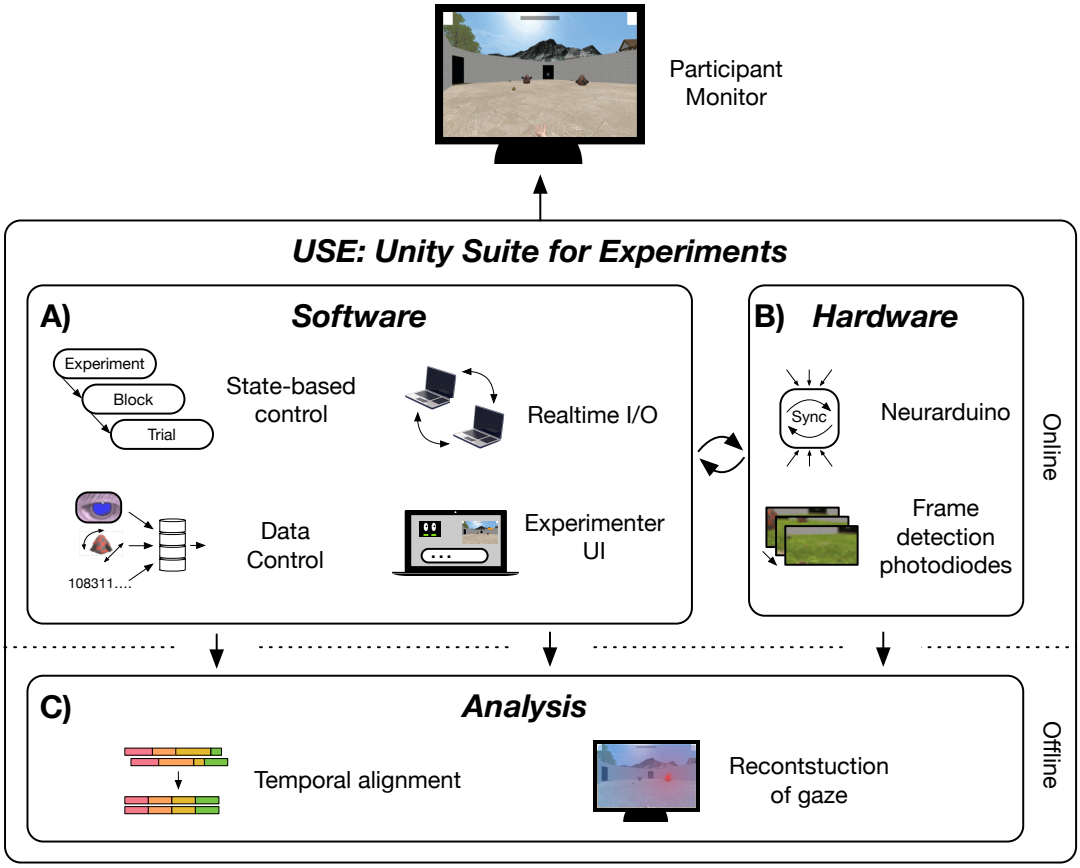

Figure 1. Unity Suite for Experiments allows strong experimental control in immersive tasks

Immersive, dynamic tasks present unique challenges to enabling strong experimental control. To address these, USE includes novel contributions for (A) software, (B) Hardware, and (C) analysis tools. (A) USE employs a novel state-based control system that underpins its strong experimental control. Realtime I/O is based on current industry standards. USE also allows experimentalists to flexibly define and collate data occurring at multiple experimental timescales, and online visualization of subject's performance. (B) The Neuroarduino is a newly developed, dedicated, and inexpensive machine that allows synchronization of different data streams (such as joystick data, eyetracker data, and game data), and can communicate with other hardware. As well, custom built photo-diode solution allows for the precise detection of physical frame onsets. (C) USE also contains a number of post-hoc analysis scripts, including a way to align all data sources, and reconstruct subject's gaze.

Figure 1 provides an overview of the three main components of USE, which are described in the remainder of this paper. A set of scripts for the video game development engine Unity 3D enables the development of nested hierarchies of experimental control, as well as tools for common experimental requirements (data collection and recording, communication with other programs and equipment, etc). USE also incorporates an Arduino-based timing and I/O hardware system to relay signals and codes between experimental hardware, and maintain a central time record. Finally, it includes a set of Matlab analysis scripts for offline data parsing, and timestream synchronization, as well as classification of dynamic gaze data and the determination of gaze targets in three-dimensional scenes.

USE allows flexible experimental protocols built around (i) dynamic or static experiments, (ii) 2D and 3D displays, and (iii) touchscreen, joystick or keyboard/button press interfaces. We anticipate that the key strength of USE is to facilitate the design and realization of dynamic studies, though it can also be used to generate static tasks (Oemisch, Watson, Womelsdorf, & Schubö, 2017; Watson, Voloh, Naghizadeh, & Womelsdorf, in press), enabling the comparison of dynamic and static task variants.

Links to the software components of USE, as well as manuals, tutorials, and example experiments, can be found on our website (http://accl.psy.vanderbilt.edu/resources/analysistools/).

2.3 Goals of USE

In developing USE, we aimed to satisfy multiple criteria. The goal was to develop a system that was:

- *temporally accurate and precise* - all data should synchronize at millisecond precision.
- *modular* - all software and hardware components are developed so that they can be used independently, or in different combinations.
- *generic* - specific components can be easily adapted to multiple purposes.
- *algorithmic* - each component of the system is explained in a principled way to facilitate implementation in other research contexts using other hardware (e.g. a different microcontroller board) or software (e.g. a different video game engine).
- *perfectly reconstructable* - every monitor frame displayed during an experiment can be recreated at will in offline analysis and combined with the synchronized information from any other data stream to reconstruct the experiment.
- *cheap* - the software is all free, the total cost of the custom hardware is under $500, and experiments can be developed and run using any modern computer and consumer-grade monitor.
- *multi-platform* - experiments can be developed on OSX or Windows computers, and run on any modern computer (including Linux), smartphone, tablet, or gaming console, and can also be developed for the web.
- *portable* - only a single computer, a small box for the arduino, two light sensors and a small number of cables are required for complete experimental control.
- *as simple and as easy to use as is feasible* - the suite solves the key challenges in dynamic experiment implementation and organization, and does so in ways that are intended to be as user-friendly as possible.
- *open source* - all components of development and analysis software and I/O data streaming should be freely available under open source licenses.

2.4 The Unity game engine

The experiment development and control software components of USE are implemented in the Unity *game engine*. Game engines are development environments for video games that

implement functionalities that games commonly require, such as the rendering of 2D and 3D graphics, physics simulation, realistic lighting, animation, sound management, etc. Unity is a free (but not open source) engine that runs on Windows, Macintosh, and Linux computers, supports building games for all major computers, phones, tablets, and game consoles, and has built-in support for stereoscopic presentation. Games made with Unity were downloaded over 5 billion times in Q3 2016 alone (Unity Technologies, 2017a). Some recent games of note made with Unity include Cites: Skylines (Colossal Order, 2015), Her Story (Barlow, 2015), Kerbal Space Program (Squad, 2015), Pokémon GO (Niantic, 2016), and Super Mario Run (Nintendo EDP, 2016). Aside from its price, Unity is attractive to small and medium-sized developers due to its relative ease of use, full feature set, comprehensive tutorials, and active online forums (Unity Technologies, 2017b). USE scripts are all written in C#, but Unity allows scripts written in either C# or Java to be used within a single project.

2.5 Specific challenges of dynamic tasks

Building an experimental suite on top of a game engine presents several unique challenges. First, game engines must ensure that all required commands are run in time for frame rendering to be carried out, but the precise execution order below the frame rate is often irrelevant, and commands can be spread across many scripts, making it difficult to follow their interactions. Command execution order in USE is guaranteed by a custom state-based system that enables the easy generation of nested hierarchies of control, such as the experiment-block-trial structure common in cognitive studies (section 3.2). Second, just as control below the level of a frame can be difficult to achieve, accurate timing below the level of a frame is, for most practical purposes, unavailable within Unity. Indeed, it is difficult even to ascertain frame onset times, complicating synchronization of displays with other experimental hardware such as neural acquisition devices. USE incorporates a novel hardware I/O solution, the Neurarduino, that acts as a central timekeeper and communication/synchronization device (section 3.3). Finally, for studies that incorporate eyetracking, identifying gaze targets in a 3D scene rendered on a 2D screen is difficult, particularly when objects have complex shapes and the subject freely moves in the 3D environment. Our solution involves a novel method named *Shotgun Raycast*, which enables the detection of all objects whose two-dimensional silhouette falls within a specified number of degrees of visual angle from a gaze point on the screen (section 3.4). These novel contributions extend Unity's robust physics and rendering capabilities, making it into a suitable platform for behavioral neuroscience research.

In the remainder of the paper we outline an experiment that provides specific examples of the various components of USE in action (3.1), before turning to an overview of USE's experimental design and control software (3.2), timing and communication hardware (3.3), and offline analysis capabilities (3.4). We then present results demonstrating the robust timing and control capabilities of USE (4), before a general discussion (5).

3 Methods

3.1 Example experiment

Static tasks built with USE have already been published (Oemisch et al., 2017; Watson et al., in press), but to demonstrate more of the suite's capabilities, we briefly present a more dynamic learning task, in which participants must navigate through a 3D scene, interact with objects, and try to learn a reward structure. A full overview of the task and results will be presented in another manuscript (Watson, Voloh, Naghizadeh, & Womelsdorf, in preparation), here we focus on reporting sufficient detail to show USE's capabilities.

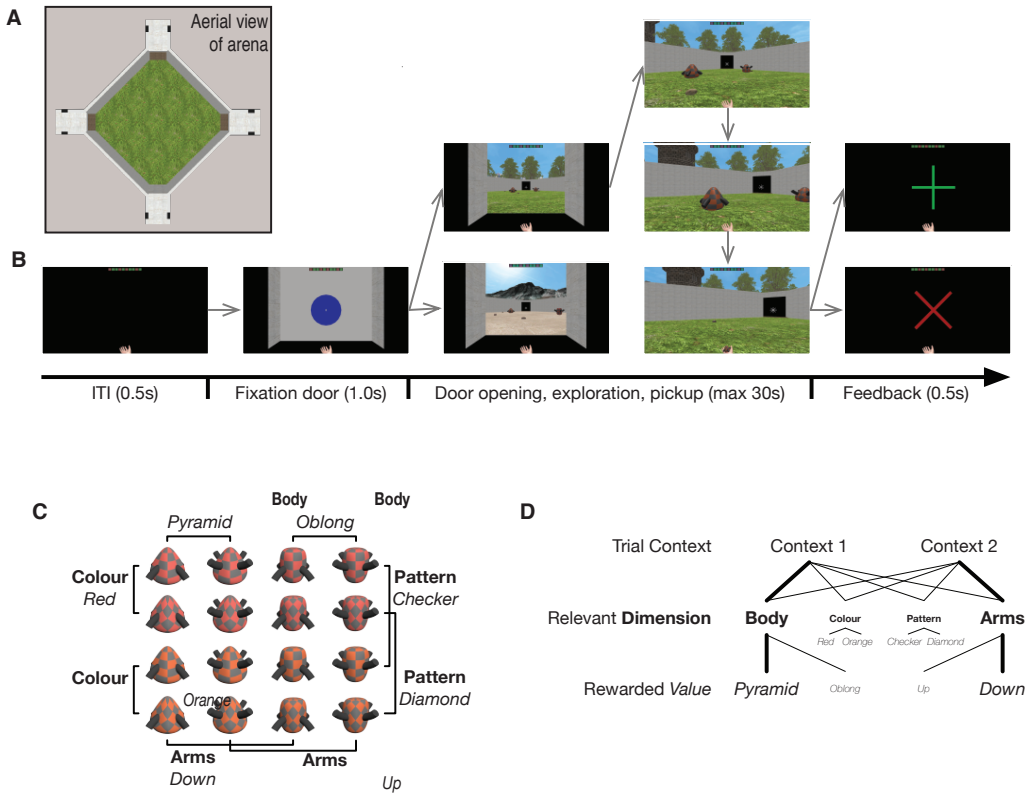

Figure 2. Context-dependent feature learning developed with USE

(A) An overhead view of the arena, participants enter through one of three corridors, and explore in the (diamond) arena. (B) An example trial sequence. Trials are separated by an ITI where the majority of the screen is black, but the avatar's hand and their reward history is visible. Participants must fixate the white dot at the centre of the blue circle in order to open the fixation door, which reveals the arena, including the context floor, and two objects (one of which is rewarded, while the other is not). Participants must then navigate towards an object and pick it up by walking over it. Finally, they take the object to one of 3 reward doors, where they are given feedback. (C) Set of quaddles used in the experiment (Watson 2018). Each quaddles is defined by four feature dimensions (body shape, color, arms position, and pattern), each of which can take on two specific values. (D) An example reward-structure that participants must learn in a given block. Objects are presented in one of two contexts. Each context has an associated relevant feature dimension, and one rewarded feature value in that dimension. In. this example, the objects presented in context 1 are rewarded if their *body* (feature dimension) is *pyramidal* (feature value). Thick lines and large font highlights the path towards the rewarded feature value, whereas small lines and font denote a path towards unrewarded values.

Figures 2a and 2b display the task environment and trial protocol. In brief, participants must choose between two objects on each trial, one of which is rewarded and one of which is not, and use trial-and-error learning to determine the context-dependent feature-based rule governing reward. There are 16 objects in total, composed of four feature dimensions of two values each (Figure 2c, cf. Watson et al., in press), and two of the feature values are rewarded, one in each of two contexts (Figure 2d). Once participants have achieved satisfactory performance on a block, new rules are generated and a new block begins.

50 participants ran in the study for approximately one hour each. Two participants were excluded due to chance performance. Gaze was tracked using a Tobii TX300 combined head-free eyetracker and OLED monitor. Figure 2e demonstrates that over the course of a block they learned to choose the target more often than the distractor, that they made these choices more rapidly, and that prior to picking up an object they made more fixations to the rewarded object

than the unrewarded one (see 3.4.2 for details of how fixation targets are determined in USE).

The Unity project containing all scripts and other files needed to build the task is available on our website.

3.2 USE: Unity Suite for Experiments

3.2.1 States and sequences

Almost all software commands in USE are controlled by an architecture of *States* and *Sequences*. This makes USE experiments a form of hierachical *finite state machine* (Wagner, Schmuki, Wagner, & Wolstenholme, 2006) whose state transitions are (usually) governed by their order in a sequence. The State-Sequence architecture is flexible enough to make it easy to generate standard experimental hierarchical relationships such as experiment-block-trial, as well as more complex structures. Here, we explain their abstract functional role and give examples of their use in the experiment described in 3.1.

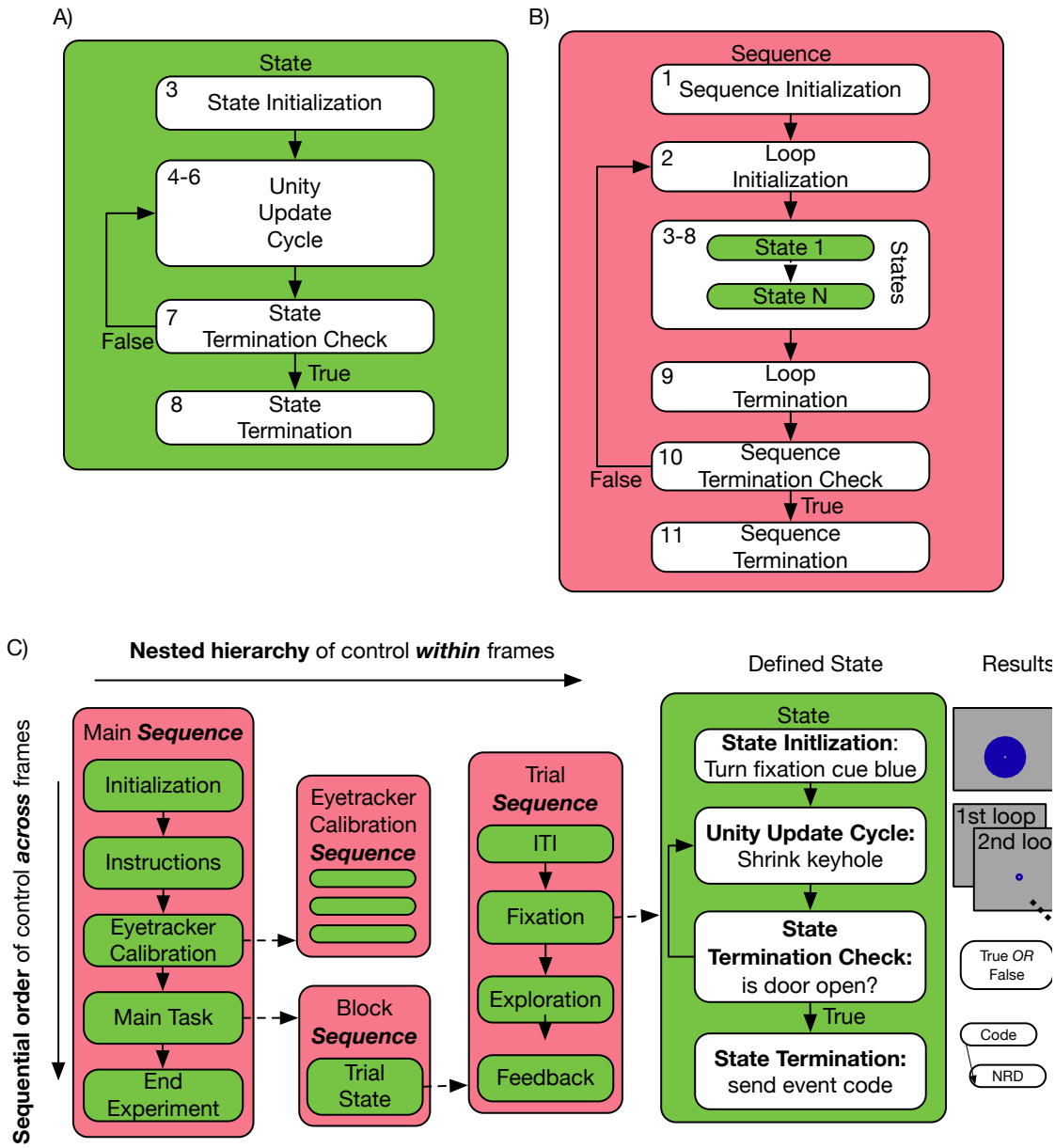

Figure 3. State/Sequence organization allows for flexible and strong experimental oversight

Depiction of (A) state and (B) sequence architecture and their interaction in (C) an example experiment. White rectangles denote method groups, and numbers in each method group denote their priority in a frame. (A) A State contains an Initialization method group that runs

once the first time a state is encountered. The Unity update cycle runs each frame until the Termination Check is True, at which point the Termination method is run. (B) A sequence defines the order of a collection of states. States can loop multiple times before a Sequence is completed. An Initialization method runs once the first time a sequence is encountered. Then, a Loop Initialization is run every time a new loop of states begins. States run consecutively as they are defined, until the Termination Check returns true, at which point the Sequence terminates. (C) Control flow of an example experiment, illustrating that sequences can be children of States. The unique setup allows nested control within frames, but also guarantees control across frames. This is only an illustrative portion of the full task hierarchy. The Main Sequence is the highest level of control. After completing an Initiation and Instruction State, the Sequence encounters a Calibration State. This State has an associated Sequence that itself has multiple states. After completing this, The MainTask state is run, which also has a nested Block Sequence, which itself has a Trial State and associated Trial Sequence. Here, we explicitly illustrate the Fixation State within the Trial Sequence, and the associated actions taken within it.

3.2.2 States: method groups defining experimental epochs

A State is an object that contains method groups defining a particular period of the experiment (Figure 3a). States are flexible and can span many hierarchies of an experiment; from experiment level controls, to instruction routines, to individual trial epochs such as inter-trial intervals, feedback periods etc. A State constitutes a number of method groups that specify commands to be run during the State, and when it starts and ends. On the first frame in which an experiment enters a new State, a `StateInitialization` method group runs. Within each frame, a number of method groups are run that control Unity's update cycle (for more details of the update cycle, see the manual on our website), which are followed by a `StateTerminationCheck` method that verifies whether an end condition for the state has been reached. If true, a final `StateTermination` method group is run on the last frame of the state, and the experiment transitions to a new State on the following frame, otherwise, the current State's update cycle methods run again on the following frame.

3.2.3 Sequences: organization of States

State-to-State transitions are (usually) governed by Sequences, which are simply containers for States that are meant to run one after the other, often in a looping manner (Figure 3b). Once one State terminates, the next State in the Sequence initializes on the subsequent frame. After the last State in a Sequence terminates, the Sequence runs its own `SequenceTerminationCheck` method group, and if this returns true the Sequence will terminate itself, or if it returns false it will loop back to its starting State.

Like States, Sequences have Initialization and Termination methods, but these are separated into two types: Loop and Sequence initialization/termination. `LoopInitialization` runs prior to the initialization methods of the first State in a Sequence, and `LoopTermination` runs after the termination methods of the last State in a Sequence. Both these methods run every time the loop starts or finishes. They are functionally equivalent to adding methods to the start of the `StateInitialization` of the first State in the sequence, and to the end of the `StateTermination` of the last State in the sequence, but explicitly segregating them in this manner makes it both easier to understand their logical role, and to avoid mistakes during development, when one often re-orders States within Sequences. Finally, `SequenceInitialization` and `SequenceTermination` methods run prior to the first `LoopInitialization` and after the final `LoopTermination` methods, respectively, but unlike these Loop methods, they only run once each, at the start of the first frame and end of the last frame in the Sequence, respectively.

3.2.4 Standalone States

USE also includes *Standalone States*, which are the same as other States, save that their transition is not governed by their position in a Sequence. They are useful for controlling events that occur irregularly during an experiment, such as gaze re-calibration, pauses, etc. Standalone States can be initialized at any time, and after their `StateTermination,` the Sequence from which they were called is either restored to its previous State, or it is reset to a different

designated State in the Sequence.

3.2.5 Hierarchical Organization of states/sequences

A critical feature of Sequences and States, central to the hierarchical flexibility of USE, is that *Sequences can be children of States*, allowing Sequences to pass control to sub-Sequences. This guarantees that commands run within a certain, experimentally defined order, protecting against potential race-conditions, and ensures that the subject's experience is as the experimenter had intended.

The State/Sequence architecture governing the example experiment (3.1) is visualized in Figure 3c. The Main Sequence is the top level of the hierarchy, which defines high-level epochs such as setup, eyetracker calibration, and the task itself. The eye tracker state has an associated Eyetracker calibration sequence, which itself has multiple states associated with it (presentation of the calibration dot, analysis of the estimated gaze location, experimenter acceptance or rejection of the calibration results, etc). One this has been completed, we progress to the Main Task state. This has embedded within it a Block Sequence, which itself passes control to a Trial Sequence. In this example, there are four trial States – inter-trial interval, fixation, exploration, and feedback. For example, the fixation State's `StateInitialization` method activates the blue fixation keyhole, its update method groups control the current size of the fixation keyhole, which shrinks as the participant fixates it and fully disappears after one second of fixation, and the current position of the door, which begins to open after the keyhole has disappeared. The `StateTerminationCheck` method simply checks if the door is fully opened, and if this is true the `StateTermination` method runs, which sends an event code to connected hardware.

Once all trials in a block have been completed, the Block Sequence's `TerminationCheck` verifies if there are blocks remaining in the experiment. If so, the Block Sequence loops back to the beginning, and a new round of trials begin, but if not, the Block Sequence ends and the Main Sequence shifts to an End Experiment State, which performs various housekeeping functions before closing the application.

We anticipate that the large majority of experiments will use at least a two-level hierarchy of Block and Trial Sequences, and that more complex experiments will benefit from a top-level Main Sequence, as used in the example experiment. State and Sequence logic provides an extremely flexible way of designing experimental hierarchies, while maintaining precise control over the order of commands during Unity's primary update loops.

3.2.6 Data Controllers

USE incorporates a generic `DataController` class that enables the flexible collection and writing of as many data streams as may be desired, and writes these to a txt file. These can include variables of any type, and each DataController object is independently controlled, meaning that data can be collected and written at different frequencies. For example, it makes sense to collect positions of the camera and moveable objects every frame, while trial accuracy might only be updated once every few seconds, and block-level information might be generated every few minutes.

Importantly for timing and later analysis purposes, in our studies we generate FrameData files, which are updated every frame and include the positions, rotations and sizes of all objects for which these values can change (see 3.4.2), as well as the current expected state of the two flickering patches beneath the light sensors used for timing alignment (see 3.4.1).

3.2.7 Experimenter View

Our laboratory typically employs USE in a two-monitor setup, one for the experimenter and one for the participant. The experimenter monitor displays three main components: (1) a panel showing a stream of exactly what the participant is viewing on their monitor, with overlaid real-time gaze positions if desired, (2) text panels displaying information of various kinds, such as messages from hardware, or summary information about participant performance, and (3)

controls to modify experimental variables (e.g. trial epoch durations). All these components are fully customizable to suit particular experimental needs.

3.2.8 Realtime I/O

USE handles real-time communications via `SerialPortController` and `UDPPortController` objects, which include methods for setting up ports, reading incoming data from system buffers, storing it in USE-specific buffers for use by other methods, and clearing these buffers. One special case of serial data are event codes, which are handled by a specific `EventCodeController`. Typically, codes are prepared a frame in advance, and sent as quickly as possible after the new frame onset. However, Unity's internal limitations mean that the latency between frame onset and event codes is less stable than might be desired, and for millisecond precision event code timestamps must be modified in offline processing (see 4.3).

3.3 The Neurarduino – timekeeping and communication

The Neurarduino is used to channel communication between experimental hardware during an experimental session, and to generate a single, highly accurate timing record that enables the time-alignment of the various data streams produced by these different pieces of hardware, as well as the physical onset of frames on an experimental display. It consists of a commercial Arduino Mega2560 r3 board with custom firmware and a custom shield connecting it to a number of I/O ports. Figure 4 shows how the Neurarduino might be used in a typical experiment.

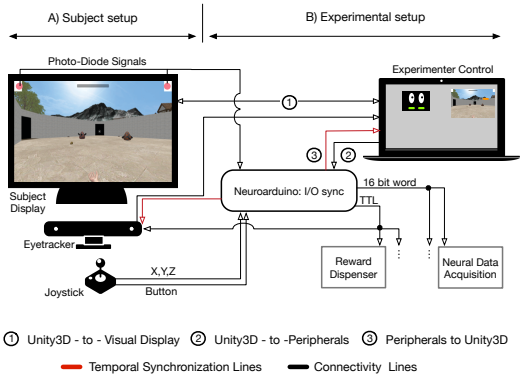

Figure 4. Example Connectivity diagram showing full capabilities of USE

A typical setup employing the full USE suite. (A) The subject setup (left) are those components of the experiment visible to the subject, including the monitor, joystick, eyetracker, and photodiodes. The subject display shows the virtual environment the subjects can interact with. They move through the environment with the joystick, and the eyetracker records their gaze behaviour. The monitor has patches in the top corners that change color to indicate a frame change, which picked up by the photodiodes. The display communicates directly with the experimental control computer. Joystick output and photo-diode signals are communicated to the Neuroarduino. (B) Th experimental setup includes the control computer, Neuroarduino, neural data acquisition device, and reward dispenser. The experimental control computer runs the experiment and sends predefined event codes to the Neuroarduino. The Neuroarduino can then forward these codes to the neural acquisition device (a 16 bit word), reward dispenser (a TTL pulse), or the eyetracker (TTL pulse). While the control computer and subject display communicate directly (1), all other communication is funneled through the Neuroarduino, wither from control computer to the peripheral device (2) or from the peripherals to the control computer (3). Black lines denote communication from one component to another, with the arrowhead showing where it inputs. Further, red lines highlight those lines emanating from the Neuroarduino that allow post-hoc temporal alignment.

The two-way connection to the experimental computer is over a USB serial port. There are eight digital I/O channels connected to BNC jacks, five analog input channels also connected to BNC jacks, and 16 digital output channels connected to a 34-pin rectangular connector. Two

of the analog inputs are dedicated to receiving data from light sensors attached to the participant monitor, which are fed through a pre-amplifier circuit which performs amplification, low-pass filtering, and DC subtraction on the light sensor signals. In the example experiment (3.1), one digital channel was used to output a timing pulse every second that was received by the Tobii eyetracker, while the rectangular port was used to send event codes to a Neuralynx Digital SX Lynx SDSata acquisition system. Full details of Neurarduino hardware are available on the USE website.

Full firmware details and code are also available on the USE website, but in brief, the Neurarduino runs an interrupt-driven loop every 0.1 ms, and a host communication loop that runs as quickly as possible, but without guaranteed timing. The host loop controls all communication over the serial port with the experimental computer, The interrupt-driven loop reads from and writes to all other inputs/outputs and maintains information about scheduled events.

The host computer can issue commands to the Neurarduino over the serial port to control most aspects of the Neurarduino's functioning, including event timing and the interval at which data is reported. The example experiment (3.1) received data with a sampling interval of 3.3 ms, and faster reports are possible. Details of the Neurarduino's timing capabilities are found in 4.1.

3.3.1 Frame detection using light sensors

To precisely track frame onsets and detect skipped frames (where an expected frame is not rendered) or stuck frames (where a single frame persists for two or more frames), we place the Neurarduino's light sensors over two patches on the experimental display. One patch changes from black to white every frame (*timing patch*), while the other encodes a binary sequence (*coding patch*). Deviations from the expected timing and sequence of blacks and whites can then be detected, and skipped or stuck frames detected (4.2). Light sensors were housed in custom 3D-printed clamps that fit a wide range of monitors (.STL files available on our website).

3.4 Analysis Pipeline

3.4.1 Time-alignment of data files

The various data files produced during an experiment are produced by devices using different clocks, which all need to be aligned to a single timestream. Currently, this happens offline. Generally speaking, experimental hardware receives timing pulses or event codes from the Neurarduino, or sends digital signals to it, making alignment a simple matter of assigning Neurarduino timestamps to each matching code or pulse, either received or generated, in the other hardware.

Alignment with data produced internally by the host computer, and with the physical state of monitor frames, is somewhat more involved. The frame-by-frame data stored during the experiment (3.2.6) includes the current status (black or white) of each of the two patches under the light sensors connected to the Neurarduino, as reported by Unity. By analyzing the sensor voltages over the course of the experiment, the physical state of these patches on each frame can be determined (see 4.2). Any skipped or stuck frames can then be identified by comparing the states of the clock and signal sensor, and the physical onsets of each frame, as opposed to Unity's estimated onset times, are determined. The FrameData files can then be modified to reflect the actual status of every single frame in the experiment. At this point, time alignment of all data is complete, as everything can now be reference to the Neurarduino's timestamps.

3.4.2 Experimental session reconstruction

During a typical experimental session, the frame-by-frame position, rotation, scale, and other properties of interest for each object in the scene is recorded in the FrameData file (3.2.6). This enables the complete reconstruction of the experimental session. Unity's physics engine is ignored entirely, and instead each object is directly assigned its properties as recorded. Each

replayed frame contains a perfect reconstruction of the three-dimensional scene, including the camera's position, and thus the image on the monitor is identical to that originally seen by the participant. In the current version of the suite, this frame-by-frame reconstruction requires custom code modifications for each experiment, a time-consuming process that we intend to streamline in the future.

The reconstruction process provides several benefits. Video of the entire experimental session, or simply particular moments of interest, becomes easy to generate offline, with no need to record during the session itself. If desired, gaze positions, mouse or touchscreen clicks, or other behavior can be overlaid on this video. We provide an video of several trials from the example experiment on our website. Individual frames can be recreated and analyzed at will, for example using saliency estimation algorithms (e.g. Borji, Sihite, & Itti, 2014). Finally, gaze targets can be determined, as described in the following section.

3.4.3 Gaze target determination

We smooth and classify gaze data using a modified version of an existing algorithm that provides superior processing of dynamic gaze data (Andersson, Larsson, Holmqvist, Stridh, & Nyström, 2016; Corrigan, Gulli, Doucet, & Martinez-Trujillo, 2017; Larsson, Nyström, & Stridh, 2013; Larsson, Nyström, & Stridh, 2014; Nyström & Holmqvist, 2010), adapted to handle noisier data by using estimated angular acceleration (Engbert & Kliegl, 2003; Engbert & Mergenthaler, 2006) and robust statistics (Wilcox, 2010). The Supplementary Methods provide details of the algorithm, which is further explained in (Voloh et al., submitted). The results of this classification are detailed in 4.4 below.

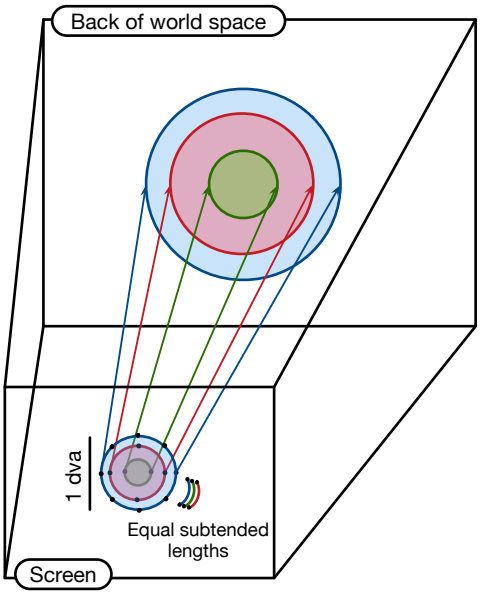

Figure 5. ShotgunRaycast allows gaze target determination in a 3D world

Moving through a 3D world means that the size and shape of the silhouette of an object on the screen constantly changes, making it a challenge to determine gaze targets. The ShotgunRaycast solves this challenge. It defines a conic section, beginning with a circle on the camera and ending with a larger circle a long distance into the world space, which has the same center on the screen and whose projection onto the screen subtends exactly the same angle. This has the effect of finding any objects whose silhouette on the screen lies either completely or partially within the circle defining the conic section. Experimentalists define the density of sampling, and radius of the circle in degrees of visual angle.

After smoothing and classification, gaze targets can be determined. For any frame of interest, the corresponding gaze points are identified, and a `ShotgunRaycast` method is used to determine which objects have been foveated (Figure 5). Building upon Unity's standard raycast function, `ShotgunRaycast` enables the specification of desired degrees of visual angle

around a gaze point, and any objects whose two-dimensional silhouette on the screen falls within this gaze point are treated as gaze targets. The density of sampling and the size of the sampled space (in degrees of visual angle) can be defined by the experimenter.

The result of this process is the sample-by-sample specification of gaze targets, which can be further analyzed according to gaze type (e.g. fixations, smooth pursuits), or objects, as required by the experiment (see 4.4).

4 Results

4.1 Neurarduino timing

The Neurarduino enables digital and analog signals to be generated with low latency and high precision (on the order of 0.1 ms for both). This is supported by a number of oscilloscope-based tests whose results are summarized below. Full details of these tests are found in the Supplementary Materials.

Precision and repeatability of digital outputs were tested by measuring timing pulses generated by the Neurarduino. Rise time and fall time were very much faster than the measurement scale of 0.01 ms, while jitter in the time between pulses was approximately +/- 0.02 ms.

Tests were also performed of the Neurarduino's ability to generate a digital signal when commanded via an external digital trigger. Rise time was again much faster than the measurement scale of 0.01 ms, while the delay between trigger and output was 0.01 to 0.11 ms for single digital outputs, and 0.10 to 0.20 ms for event code output.

Finally, the analog pre-amplifier delays light sensor signals by approximately 0.33 ms, and takes several seconds to settle after large changes to the DC light level.

4.2 Frame detection

To quantify the performance of frame detection and onset determination, signals were recorded from the clock and signal light sensors over approximately 1.5 hours. The data for each sensor was analyzed by identifying positive and negative peaks in the voltage trace rate, where intervals from positive to negative peaks were classified as black frames, while negative to positive peaks were classified as white. The median duration of black or white periods in the clock signal was 16.7 ms. In the entire test period, there were exactly 32 periods (of a total of 336,722, ~0.0001%) whose duration was not within 3.3 ms (one Neurarduino reporting interval) of 16.7, 33.3, 50.0, or 66.7 ms. Thus, effectively all white and black periods have a duration that corresponds to an integer multiple of the expected duration of a single frame on a 60 Hz monitor. Any frame where this multiple is greater than 1 constitutes a skipped or a stuck frame, which can be identified by investigating the corresponding *coding patch's* data.

4.3 Frame onset to event code timing

A critical question for many neuroscientists will be *what is the relationship between the physical onset of a frame and timing signals sent on that frame?* We tested this by connecting an oscilloscope to (a) light sensor placed on the host computer's monitor, and (b) one line of the Neurarduino's rectangular port. A USE script was written to alternate the patch of the monitor below the light sensor between black and white every frame, and to send an event code as rapidly as possible at the start of each frame.

Timing was referenced to event code output. The duration between frame onset and event code onset was stable but inconsistent. During any given stable interval, it was approximately constant with a jitter of about +/- 1 ms, but when there were disruptions to frame refreshes, and the patch did not flicker from black to white for two or more frames, the offset reset itself. Stable frame offset values were approximately 0-10 ms.

This means that offset's between frame onset and event code onset are stable over periods of tens of seconds or longer, but is not known *a priori* or repeatable between stable periods. There

is no Unity-controllable way in which this can be improved on, due to the imprecision of Unity's internal estimates of frame onset. Thus, accurate sub-millisecond time synchronization of frame onsets to event codes (and thereby the other experimental hardware) currently requires event code times to be modified in post-processing to reflect actual frame onsets, as determined in 4.2.

4.4 Gaze Classification

Gaze data from participants in the example experiment were classified using the algorithm described in 3.4.3. Table 1 shows that the velocities, amplitudes, frequencies and durations of each classified gaze period are appropriate. All these values are quite typical (Andersson et al., 2016; Nyström & Holmqvist, 2010; Otero-Milan et al., 2008). Furthermore, saccades exhibit the classic *main sequence* (Figure 6), in which velocity and amplitude are linearly related, with an inflection point occurring between 10-15° (Inchingolo & Spanio, 1985).

	Fixation	Smooth Pursuit	Saccade
Mean Velocity (deg/s)	10.1 ± 0.5 SE	16.6 ± 0.7 SE	---
Peak Velocity (deg/s)	---	---	210.0 ± 3.6 SE
Mean Amplitude (deg)	0.390 ± 0.009 SE	2.5 ± 0.07 SE	5.7 ± 0.1 SE
Rate (/s)	2.6 ± 0.07 SE	1.4 ± 0.05 SE	2.4 ± 0.07 SE
Duration (s)	0.35 ± 0.08 SE	0.32 ± 0.01 SE	0.04 ± 0.0006 SE

Table 1. Summary statistics (means and standard errors) for gaze periods classified as fixations, smooth pursuits, and saccades, averaged across all subjects performing the example experiment. The mean velocity of saccades and peak velocities of fixations and smooth pursuits are not shown, as these are irrelevant to their characterization and tend to be misleading.

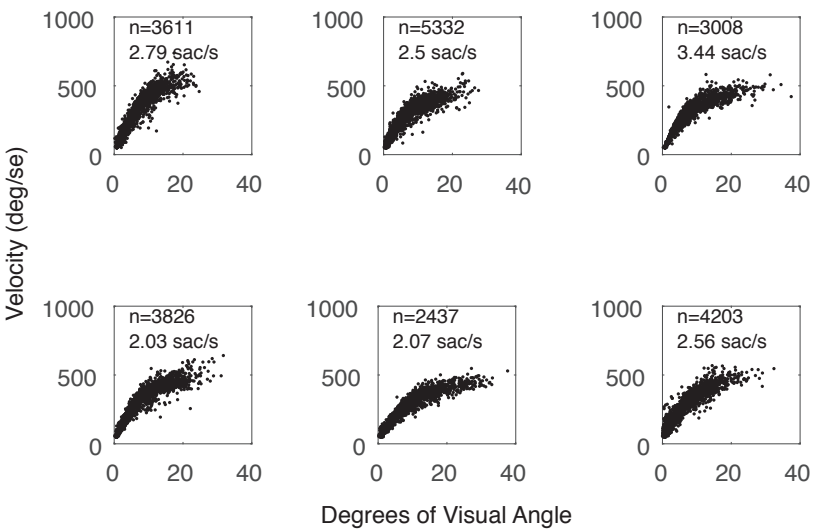

Figure 6. Saccadic main sequence for 6 example subjects

Saccadic main sequence of six representative subjects performing the context-dependent feature learning task. Subjects rate of saccades ranged from ~2-3.5 saccades per second. There is an inflection point in the main sequence that occurs around ~10-15 deg.

4.5 Fixation behaviour

After determining fixation targets using ShotgunRaycast (3.4.3), fixations to both target and distractor objects were used to generate a heatmap of fixation locations on objects, showing that participants tended to focus object centres, though there were also a smaller number of

fixations to their peripheries (Figure 7), as one might expect in such a task. Finally, participants began to preferentially fixate targets over distractors later in the block showing that their gaze behaviour reflected their rule-learning (Figure 2e), as has been shown in previous static studies of categorization learning (cf. Blair, Watson, Walshe, & Maj, 2009; Rehder & Hoffman, 2005; McColeman et al., 2014).

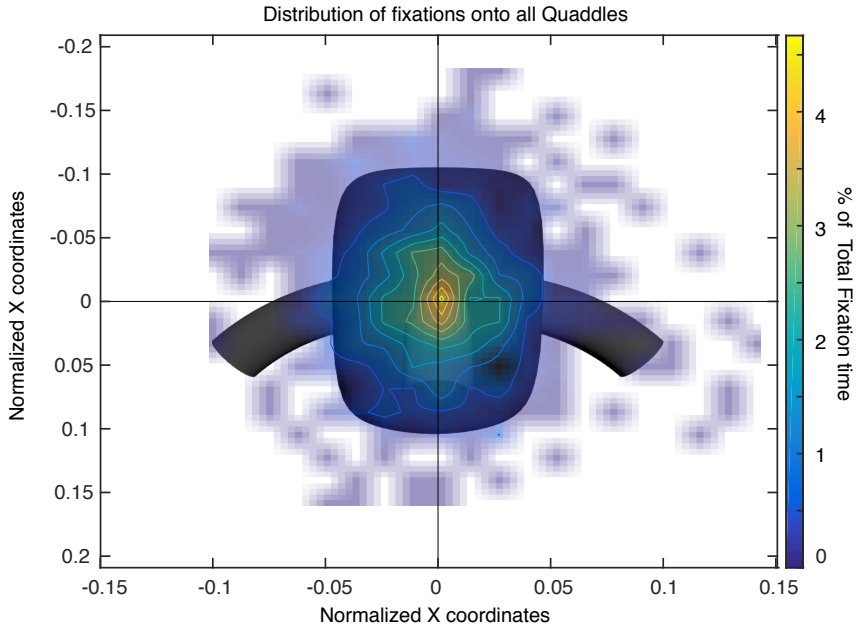

Figure 7. Reconstructed gaze shows that subjects mainly fixate the centers of objects

Density of fixations onto a standardized object for one subject (see methods). Fixation centres were normalized for change in object size with depth and mapped onto a standard object located 4 world units away from the camera. Color indicates the relative time that subjects spent fixating relative each point to the centre of the object. Subjects spent the majority of time fixating the centres of the objects.

5 Discussion

5.1 Overview

The Unified Suite for Experiments is a complete suite for the development, control, and analysis of dynamic tasks, simplifying a number of crucial challenges facing researchers interested in more realistic stimuli and possibilities of control. Its State/Sequence architecture supports nested hierarchies of control that enable the generation of tasks of any degree of complexity, the Neurarduino hardware enables reliable timekeeping and communication among experimental devices, and the offline analysis scripts enable precise time-alignment of data files and reconstruction of gaze targets.

5.2 Other software suites

Two other dynamic experimental design suites have recently been published (Doucet et al., 2016, Jangraw et al., 2014). These offer their own advantages, and we recommend that any researchers interested in dynamic tasks review them carefully. While we have an admitted bias, we believe there are several unique features of USE that may make it particularly attractive. These include:

- ***The State/Sequence architecture.*** In our experience, a specialized framework for the development of flexible experimental hierarchies greatly speeds up experimental development, and makes patterns of control more apparent.
- ***The Neurarduino.*** A generic timekeeping and communication device is a powerful tool for neuroscience research, and enables the very rapid extension of USE to any new experimental hardware without a corresponding loss of temporal precision or control.

- ***Physical frame onset detection.*** USE's ability to determine precise frame timing, and to identify which specific frames were skipped or stuck during the experiment, enables the perfect reconstruction of the experimental session, and removes the reliance on the game engine's estimates of frame onset times, which in our experience can be as much as 20 ms off the actual onset.
- ***Shotgun raycasts.*** So far as we are aware, all published work using eyetracking in 3D scenes either uses simple areas of interest, which do not change with regard to object shape, or single raycasts, which do not account for eyetracker imprecision or the spatial extent of the fovea. Shotgun raycasts, on the other hand, enable precise control over the degrees of visual angle surrounding gaze points in which objects are designated as targeted, without assuming anything about the underlying object shapes.

These advantages make USE a powerful tool for dynamic experimentation.

It should be noted, however, that the modular, generic and algorithmic principles underlying USE mean that it should be possible for interested researchers to take any of these features that are currently unique to USE, and incorporate them into other development protocols. We hope to encourage cross-fertilization of this kind, where the best development tools and concepts from different laboratories influence those in others. For example, the gaze classification algorithm we describe in 3.4.3 was adapted from that used by Doucet et al. (2016), for which we are extremely grateful.

5.3 Possible extensions

USE is robust and full-featured enough to be of use to any researchers interested in dynamic tasks. However, there are many ways in which its ease-of-use, flexibility, and power could be improved. We are currently considering several of these, including:

- Modifying the State/Sequence architecture to allow more flexible transitions between states, including multiple `TerminationChecks` that each lead to different subsequent States.
- Creating a generic event-driven data collection system for objects in the 3D environment that automatically records their location, rotation, scale, and other alterable properties whenever these properties change. This would be combined with a generic replayer system that could read in such data files and reconstruct each frame scene, without the necessity of coding a custom replayer for each study.
- Specific wrappers for communication with a large variety of experimental hardware and software (e.g. other eyetrackers, response devices, neural acquisition devices, etc).
- Real-time analysis of light sensor data in the Neurarduino, precise enough to enable signal output to be time-locked to frame onsets, and thereby enabling real-time feedback-loops used in Brain-Machine Interface contexts.

USE is under active development, and we intend to update the files available for download as components are modified or created.

5.4 Concluding remarks

The burgeoning interest in naturalistic, complex and dynamic tasks will continue to grow, and we very much look forward to seeing the effects that these tasks have on theory and method in the cognitive sciences. We hope that the set of software and hardware components we have produced will play a small part in this process, and that other researchers will begin to use USE.

6 References

Andersson, R., Larsson, L., Holmqvist, K., Stridh, M., & Nyström, M. (2016). One algorithm to rule them all? An evaluation and discussion of ten eye movement event-detection algorithms. *Behav Res Methods*. doi:10.3758/s13428-016-0738-9

Asaad, W. F., & Eskandar, E. N. (2008a). A flexible software tool for temporally-precise behavioral control in Matlab. *Journal of Neuroscience Methods*, *174*, 245-258.

Asaad, W. F., & Eskandar, E. N. (2008b). Achieving behavioral control with millisecond

resolution in a high-level programming environment. *Journal of Neuroscience Methods*, *173*(2), 235-240. doi:10.1016/j.jneumeth.2008.06.003

Asaad, W. F., Santhanam, N., McClellan, S., & Freedman, D. J. (2013). High-performance execution of psychophysical tasks with complex visual stimuli in MATLAB. *J Neurophysiol*, *109*(1), 249-260. doi:10.1152/jn.00527.2012

Barlow, S. (2015). *Her Story*. London: Sam Barlow.

Bennett, A. J., Perkins, C. M., Tenpas, P. D., Reinebach, A. L., & Pierre, P. J. (2016). Moving evidence into practice: cost analysis and assessment of macaques' sustained behavioral engagement with videogames and foraging devices. *Am J Primatol*, *78*(12), 1250-1264. doi:10.1002/ajp.22579

Blair, M. R., Watson, M. R., Walshe, R. C., & Maj, F. (2009). Extremely selective attention: Eye-tracking studies of dynamic allocation of attention to stimulus features in categorization. *Journal of Experimental Psychology: Learning, Memory and Cognition*, *35*, 1196-1206.

Borji, A., Sihite, D. N., & Itti, L. (2014). What/Where to Look Next? Modeling Top-Down Visual Attention in Complex Interactive Environments. *IEEE Trans. Syst. Man Cybern, Syst.*, *44*(5), 523-538. doi:10.1109/TSMC.2013.2279715

Bouchard, S., Guitard, T., Bernier, F., & Robillard, G. (2011). Virtual reality and the training of military personnel to cope with acute stressors. In S. Brahnam & L. C. Jain (Eds.), *Virtual Reality in Psychotherapy, Rehabilitation, and Assessment* (Vol. 6, pp. 109-128). New York, NY: Springer-Verlag, Inc.

Brainard, D. H. (1997). The Psychophysics Toolbox. *Spatial Vision*, *10*(4), 433-436.

Chaytor, N., & Schmitter-Edgecombe, M. (2003). The ecological validity of neuropsychological tests: A review of the literature on everyday cognitive skills. *Neuropsychology Review*, *13*(4), 181-197.

Colossal Order. (2015). *Cities: Skylines*. Tampere, Finland: Paradox Interactive.

Corrigan, B. W., Gulli, R. A., Doucet, G., & Martinez-Trujillo, J. C. (2017). Characterizing eye movement behaviors and kinematics of non-human primates during virtual navigation tasks. *Journal of Vision*, *17*(12), 15. doi:10.1167/17.12.15

Doucet, G., Gulli, R. A., & Martinez-Trujillo, J. C. (2016). Cross-species 3D virtual reality toolbox for visual and cognitive experiments. *Journal of Neuroscience Methods*, *266*, 84-93. doi:10.1016/j.jneumeth.2016.03.009

Engbert, R., & Kliegl, R. (2003). Microsaccades uncover the orientation of covert attention. *Vision Research*, *43*(9), 1035-1045. doi:10.1016/S0042-6989(03)00084-1

Engbert, R., & Mergenthaler, K. (2006). Microsaccades are triggered by low retinal image slip. *Proceedings of the National Academy of Sciences of the United States of America*, *103*(18), 7192-7197.

Inchingolo, P., & Spanio, M. (1985). On the identification and analysis of saccadic eye movements—A quantitative study of the processing procedures. *IEEE Trans Biomed Eng*, *32*(9), 683-695. doi:https://doi.org/10.1109/TBME.1985.325586

Jangraw, D. C., Johri, A., Gribetz, M., & Sajda, P. (2014). NEDE: an open-source scripting suite for developing experiments in 3D virtual environments. *Journal of Neuroscience Methods*, *235*, 245-251. doi:10.1016/j.jneumeth.2014.06.033

Kingstone, A., Smilek, D., & Eastwood, J. D. (2008). Cognitive Ethology: A new approach for studying human cognition. *British Journal of Psychology*, *99*(3), 317-340. doi:10.1348/000712607X251243

Kleiner, M., Brainard, D., & Pelli, D. G. (2007). What's new in Psychtoolbox-3? *Perception*,

36, ECVP Abstract Supplement.

Larsson, L., Nyström, M., & Stridh, M. (2013). Detection of saccades and postsaccadic oscillations in the presence of smooth pursuit. *IEEE Trans Biomed Eng*, *60*(9), 2484-2493. doi:10.1109/TBME.2013.2258918

Larsson, L., Nyström, M., & Stridh, M. (2014). Discrimination of fixations and smooth pursuit movements in high-speed eye-tracking data. *Conf Proc IEEE Eng Med Biol Soc*, 3797-3800. doi:10.1109/EMBC.2014.6944450

McColeman, C. M., Barnes, J. I., Chen, L., Meier, K. M., Walshe, R. C., & Blair, M. R. (2014). Learning-induced changes in attentional allocation during categorization: a sizable catalog of attention change as measured by eye movements. *PLoS One*, *9*(1), e83302. doi:10.1371/journal.pone.0083302

Niantic. (2016). *Pokémon GO*. San Francisco: Niantic.

Nintendo EDP. (2016). *Super Mario Run*. Kyoto: Nintendo.

Nyström, M., & Holmqvist, K. (2010). An adaptive algorithm for fixation, saccade, and glissade detection in eyetracking data. *Behav Res Methods*, *42*(1), 188-204. doi:10.3758/BRM.42.1.188

Oemisch, M., Watson, M. R., Womelsdorf, T., & Schubö, A. (2017). Changes of attention during value-based reversal learning are tracked by N2pc and feedback-related negativity. *Front. Hum. Neurosci.*, *11*, 7. doi:10.3389/fnhum.2017.00540

Otero-Millan, J., Troncoso, X. G., Macknik, S. L., Serrano-Pedraza, I., & Martinez-Conde, S. (2008). Saccades and microsaccades during visual fixation, exploration, and search: foundations for a common saccadic generator. *J Vis*, *8*(14), 21.1-2118. doi:10.1167/8.14.21

Peirce, J. W. (2007). PsychoPy--Psychophysics software in Python. *Journal of Neuroscience Methods*, *162*(1-2), 8-13. doi:10.1016/j.jneumeth.2006.11.017

Peirce, J. W. (2008). Generating Stimuli for Neuroscience Using PsychoPy. *Front Neuroinform*, *2*, 10. doi:10.3389/neuro.11.010.2008

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

Rehder, B., & Hoffman, A. B. (2005). Eyetracking and selective attention in category learning. *Cognitive Psychology*, *51*(1), 1-41.

Risko, E. F., Laidlaw, K., Freeth, M., Foulsham, T., & Kingstone, A. (2012). Social attention with real versus reel stimuli: toward an empirical approach to concerns about ecological validity. *Front Hum Neurosci*, *6*, 143. doi:10.3389/fnhum.2012.00143

Schmuckler, M. A. (2001). What Is Ecological Validity? A Dimensional Analysis. *Infancy*, *2*(4), 419-436. doi:10.1207/S15327078IN0204_02

Shanley, P. (2017). Why aren't video games as respected as movies? Retrieved September 6, 2018 from https://www.hollywoodreporter.com/heat-vision/why-arent-video-games-as-respected-as-movies-1067314

Slater, M., & Wilbur, S. (1997). A framework for immersive virtual environments (FIVE): Speculations on the role of presence in virtual environments. *Presence: Teleoperators and Virtual Environments*, *6*(6), 603-616.

Squad. (2015). *Kerbal Space Program*. Mexico City: Squad.

The NDP Group. (2009). More Americans play video games than go out to the movies. Retrieved September 10, 2018 from https://www.npd.com/wps/portal/npd/us/news/press-releases/pr_090520/

Unity Technologies. (2017a). Company Facts. Retrieved July 17, 2017, from https://unity3d.com/public-relations

Unity Technologies. (2017b). Learn With Unity. Retrieved July 17, 2017, from https://unity3d.com/learn

Wagner, F., Schmuki, R., Wagner, T., & Wolstenholme, P. (2006). *Modeling Software with Finite State Machines — A Practical Approach*. Boca Raton: Auerbach Publications.

Watson, M. R., Voloh, B., Naghizadeh, M., & Womelsdorf, T. (In Press). Quaddles: A multidimensional 3-D object set with parametrically controlled and customizable features. *Behavior Research Methods*.

Wilcox, R. R. (2012). *Introduction to Robust Estimation and Hypothesis Testing* (2nd ed.). Amsterdam: Elsevier/Academic Press.

Witmer, B. G., & Singer, M. J. (1998). Measuring presence in virtual environments: A presence questionnaire. *Presence: Teleoperators and Virtual Environments*, *7*(3), 225-240.

Youngblut, C. (2007). *What a Decade of Experiments Reveals about Factors that Influence the Sense of Presence: Latest Findings*. Alexandria, Virginia: Institute for Defense Analysis.