Journal of Neuroscience Methods 266 (2016) 84–93

Contents lists available at ScienceDirect

Journal of Neuroscience Methods

journal homepage: www.elsevier.com/locate/jneumeth

Research article

Cross-species 3D virtual reality toolbox for visual and cognitive experiments

Guillaume Doucet[a,c], Roberto A. Gulli[b,c], Julio C. Martinez-Trujillo[c,*]

[a] *Department of Physiology, McGill University, 3655 Promenade Sir William Osler, Montreal, QC H3G 1Y6, Canada*
[b] *Integrated Program in Neuroscience, McGill University, 3801 University Street, Room 141, Montreal, QC H3A 2B4, Canada*
[c] *Robarts Research Institute, Brain and Mind Institute, Department of Physiology and Pharmacology, Department of Psychiatry, Western University, 1151 Richmond St. N., Room 7239, London, ON N6A 5B7, Canada*

HIGHLIGHTS

- This toolbox adds VR capability to any pre-existing data acquisition framework.
- Cross-species usage, from rodents to humans, is supported.
- Possible paradigms range from simple search to complex contextual learning.
- Can be paired with eye tracking and electrophysiological recording.
- Minimizes implementation costs and doesn't require specific hardware.

ARTICLE INFO

Article history:
Received 26 January 2016
Received in revised form 10 March 2016
Accepted 11 March 2016
Available online 22 March 2016

Keywords:
Virtual reality
Virtual environment
Stimulus presentation
3D
Cognitive testing

ABSTRACT

Background: Although simplified visual stimuli, such as dots or gratings presented on homogeneous backgrounds, provide strict control over the stimulus parameters during visual experiments, they fail to approximate visual stimulation in natural conditions. Adoption of virtual reality (VR) in neuroscience research has been proposed to circumvent this problem, by combining strict control of experimental variables and behavioral monitoring within complex and realistic environments.
New method: We have created a VR toolbox that maximizes experimental flexibility while minimizing implementation costs. A free VR engine (Unreal 3) has been customized to interface with any control software via text commands, allowing seamless introduction into pre-existing laboratory data acquisition frameworks. Furthermore, control functions are provided for the two most common programming languages used in visual neuroscience: Matlab and Python.
Results: The toolbox offers milliseconds time resolution necessary for electrophysiological recordings and is flexible enough to support cross-species usage across a wide range of paradigms.
Comparison with existing methods: Unlike previously proposed VR solutions whose implementation is complex and time-consuming, our toolbox requires minimal customization or technical expertise to interface with pre-existing data acquisition frameworks as it relies on already familiar programming environments. Moreover, as it is compatible with a variety of display and input devices, identical VR testing paradigms can be used across species, from rodents to humans.
Conclusions: This toolbox facilitates the addition of VR capabilities to any laboratory without perturbing pre-existing data acquisition frameworks, or requiring any major hardware changes.

1. Introduction

In visual neuroscience, researchers have long faced the challenge of conducting ecologically valid measurements of experimental variables while maintaining strict experimental control over visual displays. For example, most visual experiments in both human and non-human primates have used simplified stimuli (e.g., bars, dots or gratings) on homogeneous backgrounds, raising the

Abbreviations: NTP, network time protocol; OOP, object oriented programming; TCP, transmission control protocol; UDK, unreal development kit.

* Corresponding author.
E-mail addresses: guillaume.doucet2@mail.mcgill.ca (G. Doucet), roberto.gulli@mail.mcgill.ca (R.A. Gulli), julio.martinez@robarts.ca (J.C. Martinez-Trujillo).

http://dx.doi.org/10.1016/j.jneumeth.2016.03.009
0165-0270/© 2016 Z. Published by Elsevier B.V. All rights reserved.

question of whether their results could be directly extrapolated to more naturalistic viewing conditions (Bohil et al., 2011; Nishimoto and Gallant, 2011). Indeed, under normal viewing conditions the retina is bombarded by a multitude of background signals and visual receptive fields seldom contain a single stationary stimulus. Although challenging, some studies have shown that it is possible to decipher basic neuronal properties (e.g., receptive field and tuning) from more naturalistic stimuli, using sophisticated analysis methods (Nishimoto and Gallant, 2011). However, ecological validity might still be undermined by the nature of the visual stimuli, often limited to passive viewing of static pictures or movies, whose relevance to the subject's natural behavior is unclear. Thus, recording and interpreting physiological signals in naturalistic environments remains a challenge for visual neuroscientists.

Modern virtual reality (VR) technology may provide a solution to this problem. It allows researchers to design and therefore strictly control dynamic, realistic and immersive environments, while closely monitoring behavioral and physiological responses during testing (Loomis et al., 1999; Bohil et al., 2011). VR technology has indeed been preferred over real stimuli to generate intuitive sensorimotor responses across multiple species, from insects to humans, as its advantages range far beyond precise experimental control (Bohil et al., 2011). Firstly, subjects are kept sufficiently static during VR navigation to enable electrophysiological or imaging experiments. Secondly, VR environments can be created, scaled and manipulated by researchers in a manner that is almost impossible in physically constrained real-world testing environments. Thirdly, VR experiments are more engaging for subjects, compared to passive viewing, as they require complex and ecologically valid behavioral responses to multisensory stimulation (e.g., approaching a virtual food source or escaping from a virtual predator). Finally, VR environments circumvent many ethical limitations by preventing injuries in "hazardous" tasks (Tarr and Warren, 2002; Slater et al., 2006; Mueller et al., 2012).

Although most drawbacks of early VR solutions (i.e., poor image quality and low level of details) have been addressed through technological advances, the resulting increase in systems' complexity and in the expertise required for their implementation have forced most laboratories to design inflexible and singularly purposed systems (Loomis et al., 1999; Bohil et al., 2011; Mueller et al., 2012; Jangraw et al., 2014). For example, laboratories interested only in monitoring behavioral responses during navigation or foraging often lack the temporal precision and/or resolution required for electrophysiological experiments (Caplan et al., 2003; Astur et al., 2004; Newman et al., 2007; Weidemann et al., 2009; Doeller et al., 2010). Additionally, previous systems have been designed in a species-specific manner by either using fixed input/output devices (e.g., gamepad or trackball) or written cues (Hölscher et al., 2005; Harvey et al., 2009; Aronov and Tank, 2014; Slobounov et al., 2015). Although many existing VR platforms are customizable to fit one's desired paradigm, they often require a two-tier architecture (i.e., one computer for the VR engine and a second computer running experimental control software). This entails learning each tier's specific script library, sometimes under multiple programming languages (Mueller et al., 2012; Jangraw et al., 2014), which greatly increases implementation cost in both time and resources. While many commercial applications have been proposed to overcome these issues (e.g., Vizard, WorldViz, USA; Eon Reality, USA), their high cost may hinder their widespread use. Furthermore, as is often the case with third-party solutions, most of commercial applications use proprietary control software and require specific input/output computer peripheral devices, which could render their implementation in an pre-existing experimental pipeline problematic (Mueller et al., 2012; Jangraw et al., 2014).

Here we aimed to create a freely available VR solution that combines professional grade graphics, high flexibility and cross-species support, which could be implemented in any existing laboratory's data acquisition framework. To achieve this, we applied the architecture proposed by Adobbati et al. (2001) and Carpin et al. (2007): remotely controlling a VR engine via simple text commands sent over a dedicated network connection. Since most programming environments implement the transmission control protocol (TCP) for network data transfer, virtually any programming language can be used to control the virtual environment (VE). Furthermore, as experimenters are most likely to select control software with which they are already acquainted, they are only required to familiarize themselves with the VR engine, greatly reducing implementation costs. As examples, we provide fully functional control script libraries based on the two most common platforms in neuroscience research: Matlab (Psychophysics Toolbox: Brainard, 1997; MonkeyLogic: Asaad et al., 2012) and Python (PsychoPy: Peirce, 2007; Vision egg: Straw, 2008). These libraries were designed to interact with the freely available Unreal Engine 3 development kit (UDK, May 2012 release; Epic Games, USA). Although UDK was specifically designed for commercial video game creation, it has been used in countless virtual applications, from static architectural design to dynamic physics simulation (e.g., driving, fire propagation). This broad range of possible applications showcases its high flexibility and ease of use, two required characteristics in any VR engines.

2. Materials and methods

2.1. General architecture

The proposed VR system completely segregates the experimenter and subject during the experimental procedures via a two-tier architecture (Fig. 1). Indeed, as information is bidirectionally exchanged between the UDK computer and the control computer, both subject and experimenter interact with their own distinct interface. The separate interfaces allow the experimenter to instantaneously modify the task parameters, while preventing input device conflicts (e.g., multiple computer mice) and preserving the subject's experience. Moreover, the available computational resources on the control computer can allow experimenters to monitor the subject's behavior by displaying position, gaze or current state information in real-time. Although it is possible to run VR experiments on a single computer, we strongly recommend to avoid non-VR operations on the UDK computer in order to optimize display quality, to prevent frame loss and maximize temporal precision. This is especially important for electrophysiological experiments where the control computer must integrate inputs (e.g., eye tracker and VR engine) and synchronize output signals (e.g., electrophysiological recording equipment and reward system) to properly guide task flow. While these procedures might not be sufficiently computationally demanding to affect display quality in purely behavioral studies, the high computing power required to render high-quality 3D environments at higher refresh rates (i.e., >100 Hz) might alter the proper timing of data recording and output signals. Lastly, as this toolbox is aimed at facilitating the addition of VR capabilities to any pre-existing visual neuroscience data acquisition framework, the simple introduction of the VR computer, while preserving the current experimental computer and its pre-existing interface with external hardware (e.g., eye tracker, electrophysiological recording system and reward system) greatly reduces implementation costs. The UDK computer and the TCP control scripts thus replace the display adapter of the previous system.

The experimental cascade begins with the subject interfacing with the UDK computer (Fig. 1, gold rectangle; 8 core 3.4 GHz Windows 7 PC with 16 GB of RAM and 2 GB of dedicated video memory) through its appropriate input device. As this framework was developed to be species independent, subjects could be rodent, monkey,

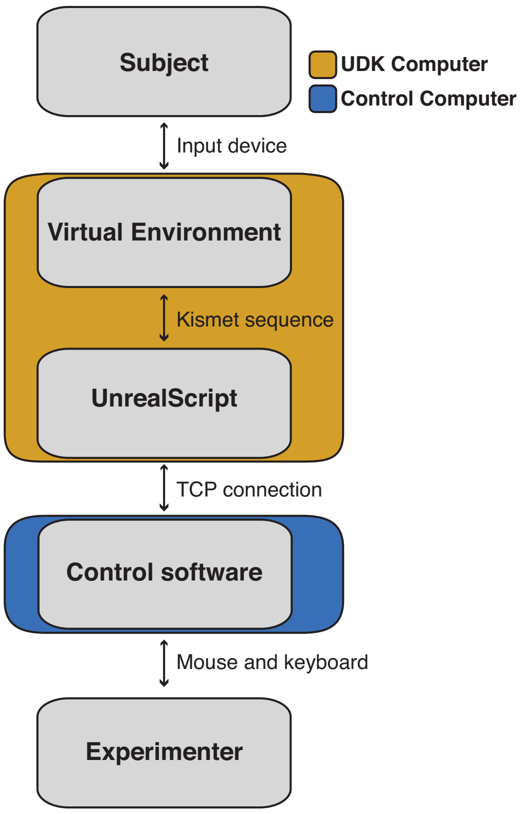

Fig. 1. Detailed system architecture. Subject actions in the virtual environments first trigger UnrealScript execution through Kismet sequences. Experimental data is then exchanged between the UDK computer (blue box) and the control computer (gold box) via a network connection. On the other hand, the experimenter can trigger changes in the virtual environment by modifying task settings in the control software. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

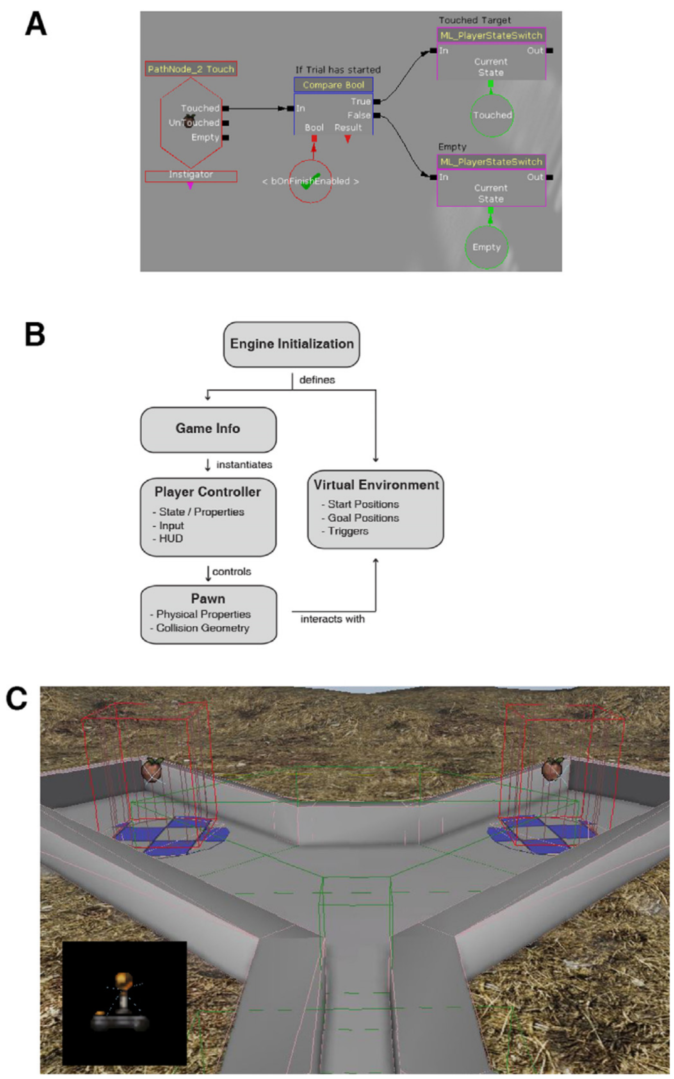

Fig. 2. Unreal Engine general architecture. A. Kismet script cascade example executed when a specific *PathNode* is touched by the subject. It then changes the current player state if the trial has started. B. Engine initialization cascade. The Game Info class defines the subclasses to use for any specific task while the virtual environment contains all the geometry and usable objects. C. Functional elements as seen in the level editor. *PathNodes* are represented by apples and their collision volume by a red wireframe mesh. *Trigger* volumes are displayed as a green wireframe mesh, *PlayerStart* as a joystick (inset) and *InterpActors* as the checkered disks.

or human. To accommodate physiological differences between species, we have created two distinct user classes: humans and animals. While humans are restricted to either a gamepad or mouse and keyboard, animals can interact with the VE through a wide variety of input devices, with the only restriction that it must be configured to operate as a computer mouse. This configuration has been widely used in the rodent VR literature, e.g., trackballs (Hölscher et al., 2005; Harvey et al., 2009; Aronov and Tank, 2014). On the other hand, many primate species have been shown to properly manipulate joysticks in virtual tasks, from capuchins (Leighty and Fragaszy, 2003) to macaques (Washburn and Astur, 2003; Sato et al., 2004; Hori et al., 2005). Although some animal studies have used gaming controllers (Kraft KC3 joystick: Leighty and Fragaszy, 2003; Washburn and Astur, 2003), we recommend the use of more recent industrial strength "plug and play" USB controllers (e.g., CTI electronics, Connecticut, USA or NSI, Belgium). However, the separation between monkeys and humans is not absolute as any joystick input can be converted to gamepad input using freely available software (i.e., x360ce, www.x360ce.com). Indeed, the following experiments in both humans and monkeys were undertaken using the aforementioned software and a modified model 215 joystick (PQ Controls, Connecticut, USA). Lastly, the range of possible user input to devices could easily be expanded beyond those typically used in electrophysiological experiments in rodents, primates and humans through vendors' drivers or device servers (e.g., haptic or position tracking devices for purely behavioral experiments such as VRPN; Taylor et al., 2001).

Following proper device configuration, the UDK computer translates user input into simulated movement in the VE. To do so, it updates the subject's previous position with the incoming input, logs the new coordinates and computes the visual scene accordingly. This is undertaken on a frame-by-frame basis. The course of the virtual simulation is in turn directed by Kismet (see Section 2.3; Fig. 2A), a visual scripting language proprietary to UDK, which acts as a two-way translator between virtual interactions and core script functions. Both Kismet nodes and core functions are written in UnrealScript, a C++ and Java inspired object-oriented programming language controlling the inner workings of the virtual engine. Kismet sequences are triggered by *Events* (Fig. 2A, red hexagon), either internally by actions in the environment or externally by commands sent by the control computer (Fig. 1, blue rectangle; 2 × quad-core 3.2 GHz Mac Pro with 8 GB of RAM and 512 MB of dedicated video memory). They can, for example, modify the variables read by the control computer when the user reaches a delimited location or, inversely, translate an experimenter's command into a visual change in the environment. Subject data from

UDK and experimenter's commands are exchanged between the two computers through a dedicated TCP connection.

2.2. Temporal synchronization

As the TCP connection is the cornerstone of the system, it is important to note that its communication speed might not be instantaneous in some programming environments. This is the case for the Matlab software, which was not designed for real-time network implementation. Yet, achieving millisecond temporal precision is of upmost importance when combining VR with behavioral or electrophysiological monitoring. According to Asaad and Eskandar (2008), true millisecond precision can only be obtained by overcoming three hurdles: synchronization of display and subject's behavior, adequately reading hardware acquisition buffers, and controlling for supra-millisecond execution speed of high order programming languages. We achieved this by first synchronizing the UDK engine to the monitor's refresh rate, allowing each displayed frame to be precisely time stamped. The UDK engine thus computes the subject's position and state for each frame and stores it in an internal buffer. Subsequently, data is read from the buffer and sent to the control computer through the network connection. However, time-stamping of subject's data on the UDK computer uses a different clock than the control computer, impairing temporal alignment. To circumvent these drawbacks, the two computers' internal clocks were synchronized with a Network Time Protocol (NTP) algorithm (Automachron freeware, http://automachron.software.informer.com/). In a small network and in optimal conditions the NTP algorithm is believed to reach sub-millisecond precision (Pitimon and Nintanavongsa, 2014). The control computer's clock is thus set as a master and the UDK computer is programmed to synchronize to it at fixed time intervals (16 s) and to log any recorded time differences.

2.3. Unreal Engine

The Unreal Engine running on the UDK computer is a highly customizable and dynamic software. Researchers and developers can interact with the engine through two open and customizable modules provided in the development kit. First, the UnrealScript programming environment, which controls the engine itself (i.e., controllers and classes). Second, the Level Editor is used to create the visual environments with their associated Kismet sequences and user interaction points (e.g., start positions, goals).

While the description of UnrealScript is beyond the scope of this paper, it is necessary to define object-oriented programming (OOP) terminology and explore a few key scripts to properly understand the engine's internal architecture. Firstly, OOP applications are modularly designed, where each module, or class, is uniquely responsible for a specific aspect of the application. An object is simply defined as a single instance of a specific class. For example, the multiple enemy objects displayed on the screen in a video game are merely individual instances of the same "enemy" class. This implies that, while they all have the same default properties upon creation, they can behave and be modified independently once instantiated. The same principles hold true for the Unreal Engine, which, upon initialization, loads a default environment, or map, and instantiates a *GameInfo* object from its class definition (Fig. 2B). The latter contains the list of specific sub-classes to use for the selected task (i.e., *PlayerController*, *Pawn* and *Server*). A single instance of the *PlayerController* class is created for each individual user inserted in the VE. Much like the strings in a puppet/puppeteer relationship the *PlayerController* acts as the intermediary between user commands and the virtual avatar: the *Pawn* class. A *Pawn* object is defined as the transient physical representation of the user in the VE. In simpler terms, it manages collisions and locomotion. Thus, if a user "dies" in a VE, the *PlayerController* object would persist and retain all static information (e.g., user identity and input type). Conversely, a novel *Pawn* object would need to be instantiated to re-initialize its transient properties (i.e., health). The same architecture holds true for the *Server* instance, which responds to connection requests by instantiating *Connection* objects. As is the case for transient *Pawns*, if a connection is lost a new and independent *Connection* object would be created upon reception of a request by the lasting *Server* object. It is through the *Connection* objects that control queries are received, parsed and executed.

Secondly, the engine needs a way for UnrealScript to directly interact with the virtual environment. This occurs through Kismet, a visual scripting language that uses linearly activated nodes to control the world's behavior (Fig. 2A). Cascades begin with *Events* ("touched" red hexagon) that are triggered by user input in the VE (e.g., *Pawn* collisions) or control queries in UnrealScript. Subsequently, either *Actions* (purple rectangles) or *Conditions* (blue rectangles), are sequentially executed until the cascade reaches an end. While *Conditions* are used to compare or evaluate *Variables* (colored circles), *Actions* directly affect the actors of the virtual scene by modifying their properties (e.g., color, state and visibility).

Lastly, functional elements need to be inserted in the environment to properly register *Pawn* behavior and initiate Kismet sequences, namely *PlayerStart*, *Triggers*, *PathNodes* and *InterpActors*. As its name implies, the *PlayerStart* (Fig. 2C, inset) indicates where the user initially appears in the environment. In our case, it is placed in a pitch-black room to display a blank screen during inter-trial intervals. *Triggers* (green wireframe box) are invisible volumes that register when the user's *Pawn* enters or exits them (i.e., collisions). For example, they could be used to precisely control when or where certain cues are displayed, by triggering separate Kismet sequences upon *Pawn* entry or withdrawal. Although *PathNodes* were initially created to guide artificial intelligence navigation across the environment, they can be configured to have a collision volume and behave like *Triggers* (apples and red wireframe volume; Fig. 2C). In our case, they mark the possible start and goal positions and, through their collision volume, indicate when the user reaches them by modifying the *PlayerController*'s state. Lastly, the engine is designed to optimize computational resources usage by pre-rendering many visual features, for instance: textures, shadows and lighting. However, in many situations, dynamic objects are required and incompatible with pre-calculated graphics. For example, if an external cue needs to be removed in one trial its shadow must be updated accordingly. *InterpActors* (Fig. 2C, checkered disks) are such objects, as they tell the engine to dynamically compute their visual properties to allow for naturalistic behavior. Since their appearance can be controlled in real-time, they are mainly used as either allocentric or contextual cues.

2.4. Control scripts

While the Unreal Engine (Kismet and Level Editor) contains information about the VE itself, it does not contain any information about trial specific objectives and conditions. Information thus needs to be exchanged between the UDK and control computers to properly execute tasks. This occurs through queries sent from the experimental control computer, to which the UDK computer replies by either modifying the environment or by sending back user data. A typical flow chart of information transmission and function execution for a single trial can be seen in Fig. 3.

2.4.1. Connect to UDK

Following engine initialization, the control computer establishes a TCP connection with the UDK *Server*.

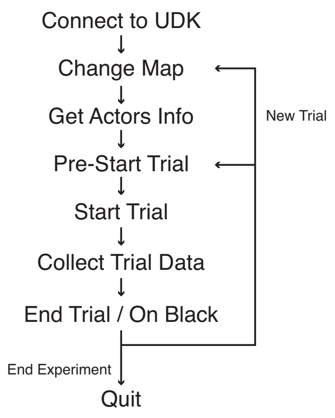

Fig. 3. Typical trial script execution cascade. Upon initialization, the control software establishes a connection with the Unreal Engine and proceeds to load the appropriate map for the selected task. It then retrieves information about the functional elements contained in the map ("Get Actors Info"). Following trial parameter selection, the control computer sends the information to the engine in the "Pre-Start Trial" phase. Data collection is initiated by trial onset, and continues until the end of trial criterion is met. Subsequent trials can either continue in the same (back to "Pre-Start Trial") or in a different (back to "Change Map") environment.

2.4.2. Change map

As an environment needs to be specified to initialize the Unreal Engine, we have programmed it to begin in a default blank environment. The control computer thus needs to send a query to load the proper map for the selected task.

2.4.3. Get actors info

When a new environment is opened, the control computer must primarily gather information about its possible targets (i.e., *PathNodes*) and cues (i.e., *InterpActors*) in order to prepare trials. This script gathers the name and position of all usable actors in the VE. The experimental software can now be programmed to either manually or randomly select the trial's start and goals positions, as well as contextual cues information.

2.4.4. Pre-start trial/start trial

This function then sends back to UDK the selected start, goals and cues, as well as their respective properties, to implement the required changes in the VE. Although the "Start Trial" script can be used to transport the subject to the selected start position and force trial onset, this procedure can also be included in the "Pre-Start Trial" Kismet sequence, leaving user initiated movement as the trial start trigger. In this scenario, the trial start would be time-stamped by UDK through a change in the *PlayerController*'s state, later read by the control software.

2.4.5. Collect trial data

While most data collection occurs during the active trial phase, it can also be used to monitor the subject's behavior during inter-trial intervals. It could thus prevent a new trial from starting while keyboard keys or joystick are engaged, enforcing the release of input devices between trials. Regardless of trial epoch, the data received from UDK contains, for each displayed frame since the last query, the time-stamped player positions, rotations and states (i.e., last touched *Trigger*/*PathNode* or trial initialization time).

2.4.6. End trial/on black

By analyzing the acquired list, the control software can compare either user position or state values with the previously selected goals and evaluate if a trial termination criterion has been met. Subsequently, an "End Trial" command could be issued to reset Kismet variables and inactivate *PathNodes* collisions in preparation of the next trial. Alternatively, an "On Black" command could be sent to transport the user back to the *PlayerStart* position to have a blank inter-trial interval screen. Though these two functions could be used interchangeably and seem redundant, their combination permits the introduction of a "free roam" period, in which the subject can freely explore the environment for a limited time at the end of each trial.

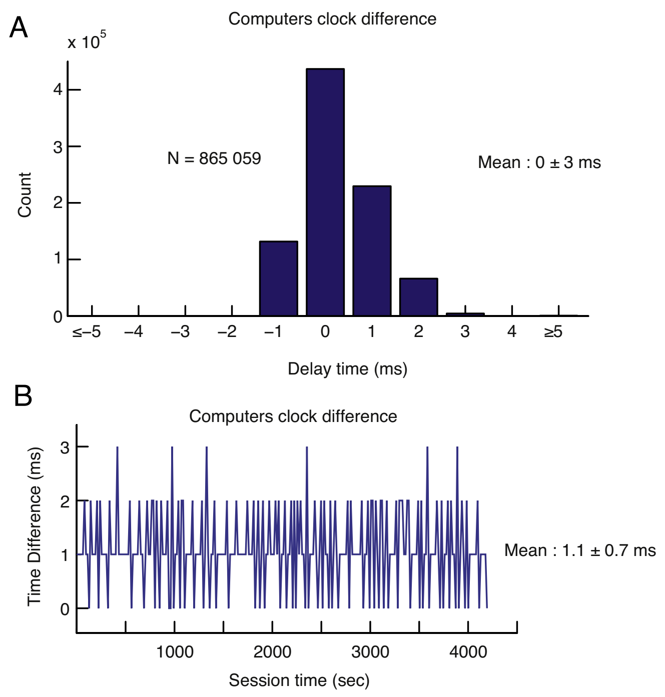

Fig. 4. Performance of the clock synchronization algorithm. A. Distribution of time differences between the Unreal and control computers as recorded by the algorithm over a 5 month period. B. Recorded time differences between computers during a ~1 h recording session, under normal experimental conditions and heavy network load. Values indicated are mean ± standard deviation.

In summary, following complete engine initialization (Fig. 2B), the execution of the "Connect to UDK" control function (Fig. 3A) results in the instantiation of a new *Connection* object by the UnrealScript *Server*. Queries (e.g., Change Map or Get Actors Info) can then be received and parsed to execute the appropriate UnrealScript functions. These functions are either activating Kismet sequences (e.g., Pre-Start Trial or On Black) to change the VE or reading *Pawn* collision data (i.e., states) stored in the *PlayerController*'s buffer. These data are then parsed into a single string and sent back to the control computer for analysis.

3. Results

3.1. Clock synchronization

As previously stated, the NTP algorithm is believed to allow for sub-millisecond temporal precision in an optimal setting. However, these values might vary greatly under normal operating conditions. We thus aimed to quantify the efficiency of the synchronizing algorithm by evaluating the reported time differences between the two computers' clocks. The Automachron software (http://automachron.software.informer.com/) is programmed to synchronize clocks once every 16 s and to log the resulting time difference. By analyzing the logged values for a period of over 5 months, we found an averaged difference value of 0 ± 3 ms (mean ± standard deviation; Fig. 4A). Since most of the recorded values occurred while the computers were unused, the relatively high standard deviation and the presence of extreme values (≤−5 ms and ≥5 ms)

Table 1
Script execution and communication timing.

Sending query in Python	Receiving query in UDK	Executing query in UDK	Receiving data in Python	
<1 ms	2 ± 1 ms	<1 ms	<1 ms	N = 4000
Sending query in Matlab	Receiving query in UDK	Executing query in UDK	Receiving data in Matlab	
<1 ms	4 ± 4 ms	<1 ms	24 ± 11 ms	N = 4000

could be explained by impaired synchronization during experiments' high network load. To test this hypothesis, we recorded an hour-long session of a subject undertaking the contextual learning task, described in Section 3.2.2. Fig. 4B shows that synchronization is only slightly affected by experimental procedures as the mean delay value barely changed (1.1 ± 0.7 ms; N.S. two-sample *t*-test). Considering the rare occurrence (<1%) of extreme delays and the robustness of the algorithm during heavy load, we thus estimated the temporal jitter of the synchronizing algorithm under any load to be ±3 ms.

It is of note to mention that the clock synchronization algorithm is only necessary for non-real-time network implemented programming environments. Having developed the control script library in real-time and non-real-time enabled languages, Python and Matlab respectively, further enabled us to validate the synchronizing algorithm's precision and to quantify Matlab's performance in TCP data transfer. To do so, we sent a total of 4000 "Collect Trial Data" queries, on two separate sessions and under normal experimental conditions, while precisely time-stamping every step of the process. As the first row of Table 1 shows, Python's real-time capabilities were confirmed considering that the total time for data collection fell below 1 ms. Furthermore, when we compared the timing values between Python and UDK ("Receiving query in UDK" column), the resulting difference of 2 ± 1 ms (mean ± standard deviation) concurred with our previously measured synchronization jitter. On the other hand, Matlab's performance was far from optimal with a total time of 24 ± 11 ms. Although the synchronization algorithm prevents issues with temporal alignment of data acquired from Matlab, it greatly impedes the rate at which it can be sampled. Indeed, control software written in Matlab would fail to sample subject data at more than 25–30 Hz and would introduce a small, albeit barely noticeable, ∼25 ms delay between environmental events and control software response.

3.2. Example tasks

3.2.1. Morris water maze analogue and radial arm maze

As basic examples we provide complete implementation of two frequently used cognitive tasks, namely the Morris water maze (Morris, 1984) and the radial arm maze (Olton and Samuelson, 1976). Since these tasks, or analogous derivatives, have been widely used across species, in both virtual and real-world situations (D'Hooge and De Deyn, 2001; Washburn and Astur, 2003; Astur et al., 2004; Hori et al., 2005; Bohil et al., 2011), they are the best example to showcase our system's simplicity.

General layout of the environment for both tasks is illustrated on the left of Fig. 5A and B, while example screen captures are shown on the right. Initial *PlayerStart* (joystick image) is located in the small square left of the environment (i.e., pitch black room). Since is it located below the main arena, it is invisible to the subject as depicted by the screen captures. In both tasks, *PathNodes* can be interchangeably used as start or goal location and could either be constant or randomized across trials. Of note, *PathNodes* being invisible in the VE, goal locations must then be identified by using colored (white) disks in the Radial Arm paradigm (Fig. 5A right). *InterpActors* (purple circles, left; red and yellow spheres, right) are used as external cues. While we have used spheres for simplicity, all displayed items can be entirely customized in both color and form.

3.2.2. Contextual learning task

At the other end of the spectrum, we have developed a more complex contextual learning task using the proposed architecture, by simply modifying a few scripts and Kismet nodes. Furthermore, this task utilizes the continuous navigation option, which means that completed trials are presented in an uninterrupted fashion, with no inter-trial intervals. The only exception occurs if a subject fails to reach his target in the allocated time. Considering that the next trial could start with the user in an inappropriate position, the subject is sent back to the *PlayerStart* (i.e., blank screen) for a short period as "penalty" for not completing the trial properly, before being sent back in the arena at the predetermined start position.

The arena has been created as a double-ended Y maze, in which the subject navigates back and forth along the north-south axis to complete trials (Fig. 6A). This task uses two contexts that define the symmetry of a three-level color-defined reward hierarchy (Fig. 6B). The largest rewarded color in one context is thus the least rewarded one in the other, and vice-versa. While the two possible contexts are constant across sessions, the rewarded color hierarchies must be re-learned each session through trial and error. Fig. 6C and D illustrates trajectory examples from two trials (C) and overall learning performance (D) of a macaque monkey performing the task. A typical trial would begin exactly where the previous one ended, with current goals and contextual information hidden (beginning of solid black arrow, Fig. 6C). The subject must then turn around and navigate to the central corridor (section between lines A and B). Upon entrance, contextual information is revealed on the inside walls of the arena. Similarly to the central corridor, entry in the decision area triggers the simultaneous appearance of both goals (line B for trial 1 and A for trial 2). The subject must then gather information about the presented colors and their associated context, and navigate to the disk that maximizes reward (white arrow, Fig. 6B bottom). Subsequently, the areas' organization is flipped along the north-south axis, transforming the decision area into the start area of the next trial.

Understandably, to accommodate the current task requirements, few basic scripts needed modifications in UnrealScript. However, from the list presented in Fig. 3A, only the "Pre-Start Trial" script was altered to include information about the trial direction (north or south), the selected context, and their associated goals (color and position). The remaining differences from the simpler task presented above came from the addition of *Triggers* and custom Kismet nodes to control context and goals appearance. Surprisingly, the largest amount of work was required on the control computer side to properly select the trial/goals combination, to validate selected target and reward the subject accordingly.

4. Discussion

In summary, the proposed toolbox was designed to facilitate the introduction of VR capabilities to pre-existing visual neuroscience data acquisition systems. To do so, it responds to simple texts commands received over a TCP connection, which can be sent by virtually any programming language. Implementation costs

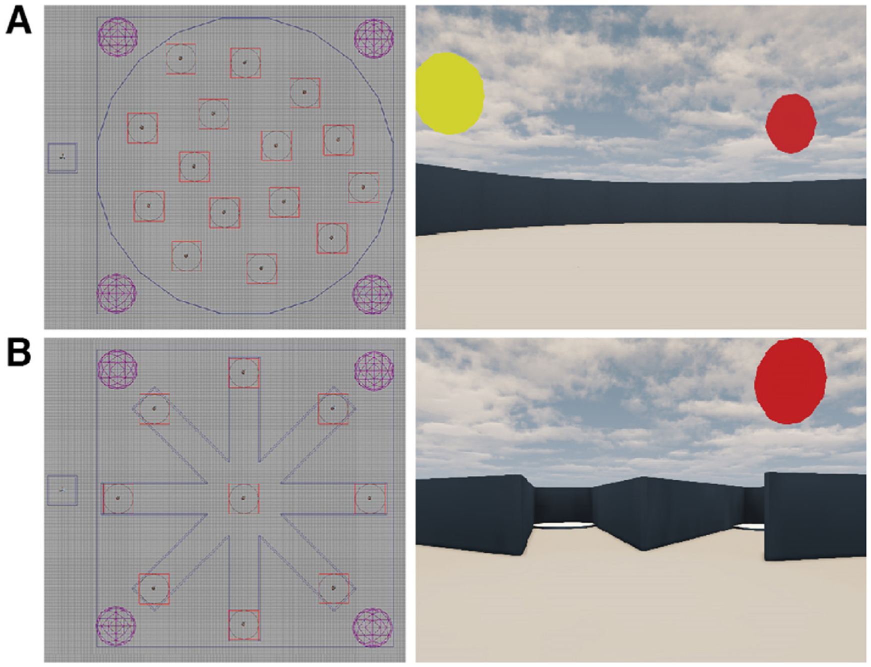

Fig. 5. Overhead views and screenshots for the Morris water maze analogue (A) and the radial arm maze (B). LEFT: general layout of the virtual environment. The *PlayerStart* is located on the left (joystick in square), walls are illustrated as blue lines, *PathNodes* as apples inside the circle contained in the red squares and allocentric cues (*InterpActors*) as purple circles. RIGHT: Examples views of the virtual environment as would be seen by subject.

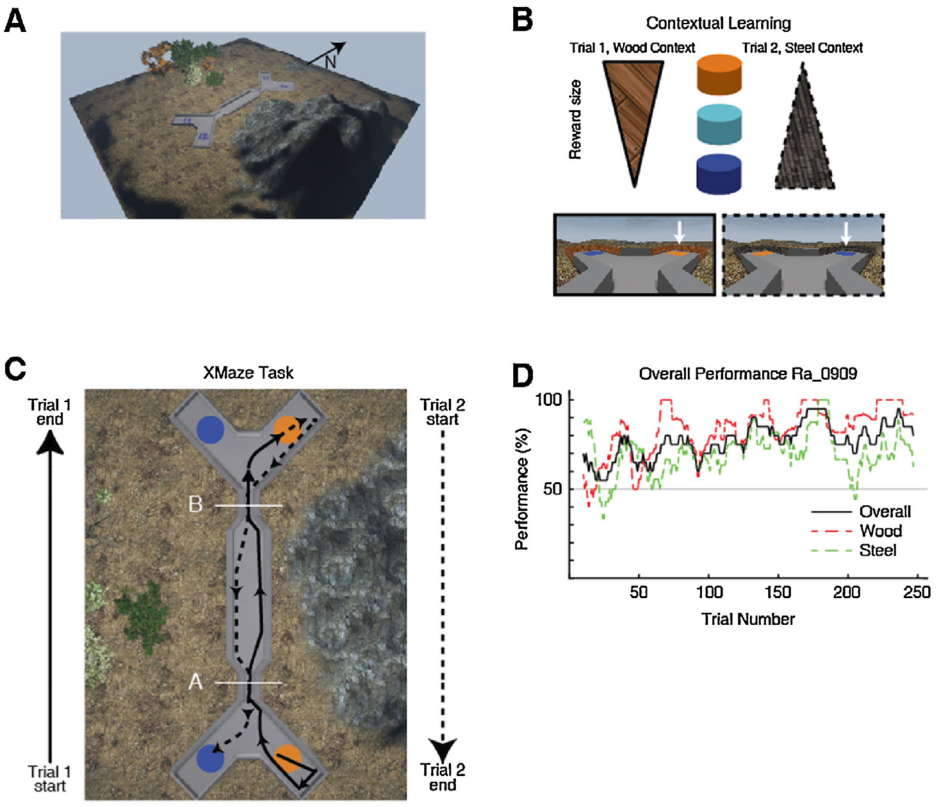

Fig. 6. Overview of the contextual learning paradigm. A. Overview of the virtual environment. The "north" end of the maze is identified by the black arrow. B. Reward hierarchy is dependent on contextual information where the highest reward in one context (e.g., orange color in Wood context) equates to a null reward in the second context (e.g., Steel context). Color hierarchies were randomly generated on a daily basis. C. Trajectory examples for two trials (solid and dashed lines) undertaken by a macaque monkey during task training. All trials begin where the previous trial ended. Contextual information then becomes available upon entry in the central corridor (A for trial 1 and B for trial 2). Continuing navigation into the decision area then reveals the trial specific goals (B for trial 1 and A for trial 2), to which the subject navigates, thus ending the trial. D. Single session training performance in a 20 trial sliding window from monkey Ra. Individual performances in each context are illustrated by the colored dashed lines. Although the monkey gets a small amount of juice reward for selecting the middle color (cyan), trials are considered "hits" only when the monkey selects the color associated with the highest reward.

are thus minimized by preserving the prior system's architecture, not requiring any hardware change to interface with external devices (e.g., eye tracker or reward system), and by allowing experimenters to utilize programming environments with which they are already familiar. Furthermore, it provides experimenters with key functional elements (e.g., Kismet nodes) to create a broad range of cognitive paradigms with minimal, if any, programming requirements. Lastly, it addresses several drawbacks from previously proposed VR solutions, namely: single species support, insufficient temporal resolution for electrophysiological experiments, the requirement for multiple external libraries and having to learn multiple programming languages to operate.

4.1. *Cross-species support*

Although this toolbox was developed and tested only on human and non-human primates, the presented VR engine is flexible enough to be used with any species capable of VE navigation. Indeed, it can be configured to address several potential shortcomings of using the toolbox with non-primate species: namely, physiological differences in size, perception and cognitive abilities.

Firstly, while flat computer monitors are sufficient for animals with frontally positioned eyes and stereovision (i.e., primates), they are not ideal for species with laterally positioned eyes (e.g., birds or rodents). To circumvent this problem, the VR engine's displayed field of view is configurable to accommodate cylindrical or large format displays. It can further be configured to output a stereoscopic signal either through *RealD* 3D side-by-side display (RealD Technologies, CA, USA), NVidia's 3D Vision (NVidia Corporation, CA, USA) or AMD's HD3D (Advanced Micro Devices, Inc., CA, USA), without external plug-ins or additional drivers, unlike most competing engines (Jangraw et al., 2014).

Secondly, since visual aptitudes such as colored vision and contrast sensitivity can vary greatly across species, experimenters can rapidly change the appearance of a visual scene through simple texture manipulation or "post-process" effects (e.g., color desaturation or visual distortion).

Lastly, physical restrictions must be addressed when selecting proper subject interface device and when defining its avatar in the VE. Indeed, small rodents and larger primates differently experience the same environment due to variations in height and walking speed. Both of which can be modified in the *Pawn* properties. Furthermore, the movement speed of the avatar is directly linked to the type of input device used. While any input device configured as a mouse (i.e., joystick, trackball, treadmill) can be used with this toolbox, each one would need to be specifically calibrated with the *Pawn*'s speed property. This process is easily undertaken as 1 unit of length in the VE is typically equivalent to 2 cm in the real world.

4.2. *Clock synchronization and temporal precision*

While the toolbox was designed to study high-order cognitive processes, it could also be used for lower-level studies. Unfortunately, the temporal precision achieved by the NTP algorithm, correcting poor TCP performance in non-real-time environments, might not be sufficient for such low latency studies. Many alternatives to improve synchronization performance and limit jitter are now available to researchers.

Firstly, a more efficient standard for time protocol algorithms, believed to reach sub-microsecond precision, has been recently developed (Precision Time Protocol, IEEE Standard 1588-2008). Although most implementations of this algorithm on the Windows platform are only available as commercial products (Domain Time II, Greyware Automation Products Inc, Texas, USA; Real-Time Systems GmbH, Real-Time-Systems, Germany), their low cost and provided customer support renders them easily accessible. On the other hand, open-source alternatives are available on the Linux and Apple platforms (Precision Time Protocol Deamon, http://ptpd.sourceforge.net).

Secondly, most programming environments can benefit from using lower-level languages libraries. For example, Matlab can interface with C/C++ or FORTRAN languages through the "mex" functions. It is then possible to use these languages' real-time TCP capabilities to avoid any TCP communication delays. This will be undertaken in further versions of this toolbox.

Thirdly, we emphasised temporal precision and encouraged using two separate computers to maximize the frame rate of the UDK computer, in order to ultimately reduce display latency. This phenomenon, defined as the delay between user action and change on the visual display, is a major issue in immersive VR technology as it can generate motion sickness, especially in head-mounted displays, but is frequently disregarded in visual neuroscience. This latency can come from three separate sources: input devices, VR software and display devices. A total latency of less than 20 ms is typically recommended to avoid motion sickness (Zheng et al., 2014). We strongly recommend caution in choice of the display device, since modern input device latency is often negligible and the engine's refresh rate is synchronized with the display device. Indeed, LCD monitors are generally slower than older CRT monitors as they pre-process the image before displaying it and their input latency has been shown to range from 1 ms to 150 ms for older monitors (Garaizar et al., 2014). Display latencies less than 20 ms can thus be achieved using monitors with >100 Hz refresh rates and low-input latencies.

4.3. *Current and newer versions of Unreal Engine*

While there are many other video game engines available (Quake 2: Harvey et al., 2009; Unreal Engine 2: Doeller et al., 2010; Vision Engine: Mueller et al., 2012; Ogre3D: Ravassard et al., 2013; Unity3D: Jangraw et al., 2014), we have found UDK (May 2012 release) to provide the best combination of power, flexibility and intuitive use. Unlike its nearest competitors (i.e., Unity and Panda3D), it does not require external libraries to present multisensory stimuli or to interface with stereoscopic displays (Jangraw et al., 2014). Furthermore, it contains a broad library of scripts, 3D models, textures, particles and sound effects, reducing the need for content creation. Lastly, since it was launched over 10 years ago and used in more than 100 professional video games, it has a remarkable level of developer support through thousands of lines of codes readily available to implement drastic engine changes (UDK Gems, https://udn.epicgames.com/Three/DevelopmentKitGems.html), and extensive documentation, libraries and bug reports from the Epic Game's user forums (https://forums.epicgames.com/forums/366-UDK). Moreover, unlike Unity, in which users need to filter the documentation across three different programming languages, UDK's single programming language renders all available tutorials relevant.

While undertaking these experiments, a newer and more powerful version of the Unreal Engine (version 4), was freely released to universities (September 4th 2014) and to the public (March 2nd 2015). Due to a major reconstruction of the engine's internal architecture, the present work could not be transferred to this environment prior to publication. This is unfortunately a recurring event, since most neuroscience laboratories developing VR applications are faced with very limited resources. Indeed, it is not uncommon that at the time of publication of a study, a newer version of the proposed video game engine is already available. For example, Harvey et al. (2009) published results using the Quake II engine four years after the release of the Quake III engine. The same holds true for Doeller et al. (2010) using the Unreal Engine 2, 6 years following the availability of Unreal Engine 3.

Understandably, small laboratories cannot keep up with multinational private companies and delays between novel technology development and its application in a research setting are expected. As video game companies only have to deal with technological considerations, laboratories must go through a complex iterative process of task design, implementation and preliminary data collection before undertaking finalized experiments. However, this delay can be significantly shortened by facilitating implementation to replicate and expand experiments. For this reason, the entire libraries of Matlab, Python and UnrealScript scripts, as well as documentation, will be made available for download at http://martinezlab.robarts.ca/, while work is carried out to transfer this environment onto a new generation engine.

4.4. Matlab and Python architectures

Matlab has been widely used in neuroscience research for decades. Through its extensive toolbox selection, ranging from robotics and computer vision to machine learning and statistical analyses, it can undertake any data processing task. Furthermore, it benefits from a wide variety of third-party toolboxes that tackle every step of an experiment, from stimulus presentation and data collection (Psychophysics Toolbox: Brainard 1997; MonkeyLogic: Asaad et al., 2012) to the analysis of any possible signals (electrophysiology: Chronux toolbox, Bokil et al., 2010; functional imaging: TDT, Hebart et al., 2015; microscopy: Patel et al., 2015). Lastly, as it has been used for many years, textbooks (Wallisch et al., 2009) and online courses have been developed around it.

On the other hand, recently emerging from the modelling field, Python is now increasingly used across all neuroscience fields. Although its current form was only developed in the late 1990s/early 2000s, this open-source software is rapidly evolving. Indeed, the list of available toolboxes (described in Muller et al., 2015), although not as extensive as Matlab's, now covers most experimental steps from stimuli generation to complex systems' analysis. Additionally, a novel programming interface, IPython notebooks, is reinventing the way scripts are executed and shared (Shen, 2014). In this environment, descriptive texts, scripts and results are intertwined in a single user-friendly interface in order to facilitate communication and reproducibility (Topalidou et al., 2015).

Since both languages are widely used and offer comparable processing power, we did not select one over the other for this study. While Matlab presents poor TCP performance, it compensates with a unified architecture, massive help documentation and customer service. On the other hand, Python might benefit from open-source development, improved sharing capabilities through IPython notebooks and high flexibility (Perkel 2015); it greatly suffers from the slow transition between two incompatible versions (2.7 and 3.4). Furthermore, by selecting only one of the two proposed programming languages, we would have greatly reduced the usability of this toolbox. Indeed, even if Python presents better performance in this context, the temporal cost of switching an already established Matlab based experimental pipeline to a Python one would greatly surpass any obtained benefits. We have thus developed and will continue to support both programming languages.

Acknowledgements

This work was supported by Canadian Institutes of Health Research, Natural Sciences and Engineering Research Council, and Canada Research Chair program grants to J.M.-T. R.G. was supported by a Natural Sciences and Engineering Research Council Post-Graduate Scholarship and a McGill University David G. Guthrie Fellowship in Medicine. We thank Mr. Walter Kucharski and Mr. Stephen Nuara, for technical support; and members of the J.M.-T. Laboratory for providing writing assistance and proof-reading this manuscript.

References

Adobbati, R., Marshall, A.N., Scholer, A., Tejada, S., Rey, M., Kaminka, G., et al., 2001. Gamebots: a 3D virtual world test-bed for multi-agent Research. Proc. Second Int Work Infrastruct. Agents MAS Scalable MAS 45, 1–6.

Aronov, D., Tank, D.W., 2014. Article engagement of neural circuits underlying 2D spatial navigation in a rodent virtual reality system. Neuron 84 (2), 442–456.

Asaad, W.F., Eskandar, E.N., 2008. Achieving behavioral control with millisecond resolution in a high level programming environment. J. Neurosci. Methods 173 (2), 235–240.

Asaad, W.F., Santhanam, N., McClellan, S.M., Freedman, D.J., 2012. High-performance execution of psychophysical tasks with complex visual stimuli in MATLAB. J. Neurophysiol. (October 2012), 249–260.

Astur, R.S., Tropp, J., Sava, S., Constable, R.T., Markus, E.J., 2004. Sex differences and correlations in a virtual Morris water task, a virtual radial arm maze, and mental rotation. Behav. Brain Res. 151 (1–2), 103–115.

Bohil, C.J., Alicea, B., Biocca, F.A., 2011. Virtual reality in neuroscience research and therapy. Nat. Rev. Neurosci. 12, 752–762.

Bokil, H., Andrews, P., Kulkarni, J.E., Mehta, S., Mitra, P.P., 2010. Chronux: a platform for analyzing neural signals. J. Neurosci. Methods 192 (1), 146–151.

Brainard, D.H., 1997. The psychophysics toolbox. Spat. Vis. 10 (4), 433–436.

Caplan, J.B., Madsen, J.R., Schulze-Bonhage, A., Aschenbrenner-Scheibe, R., Newman, E.L., Kahana, M.J., 2003. Human theta oscillations related to sensorimotor integration and spatial learning. J. Neurosci. 23 (11), 4726–4736.

Carpin, S., Lewis, M., Wang, J., Balakirsky, S., Scrapper, C., 2007. USARSim: a robot simulator for research and education. Proc. IEEE Int. Conf. Robot. Autom. 2007, 1400–1405.

D'Hooge, R., De Deyn, P.P., 2001. Applications of the Morris water maze in the study of learning and memory. Brain Res. Rev. 36, 60–90.

Doeller, C., Barry, C., Burgess, N., 2010. Evidence for grid cells in a human memory network Christian. Nature 463 (7281), 657–661.

Garaizar, P., Vadillo, M.A., López-de-Ipiña, D., Matute, H., 2014. Measuring software timing errors in the presentation of visual stimuli in cognitive neuroscience experiments. PLoS One 9 (1), e85108.

Harvey, C.D., Collman, F., Dombeck, D.A., Tank, D.W., 2009. Intracellular dynamics of hippocampal place cells during virtual navigation. Nature 461 (7266), 941–946.

Hebart, M.N., Görgen, K., Haynes, J.-D., 2015. The Decoding Toolbox (TDT): a versatile software package for multivariate analyses of functional imaging data. Front. Neuroinf. 8, 88.

Hölscher, C., Schnee, A., Dahmen, H., Setia, L., Mallot, H.A., 2005. Rats are able to navigate in virtual environments. J. Exp. Biol. 208 (Pt. 3), 561–569.

Hori, E., Nishio, Y., Kazui, K., Umeno, K., Tabuchi, E., Sasaki, K., et al., 2005. Place-related neural responses in the monkey hippocampal formation in a virtual space. Hippocampus 15 (8), 991–996.

Jangraw, D.C., Johri, A., Gribetz, M., Sajda, P., 2014. NEDE: an open-source scripting suite for developing experiments in 3D virtual environments. J. Neurosci. Methods 235, 245–251.

Leighty, K.A., Fragaszy, D.M., 2003. Joystick acquisition in tufted capuchins (*Cebus apella*). Anim. Cogn. 6 (3), 141–148.

Loomis, J.M., Blascovich, J.J., Beall, A.C., 1999. Immersive virtual environment technology as a basic research tool in psychology. Behav. Res. Methods Instrum. Comput. 31 (4), 557–564.

Morris, R., 1984. Developments of a water-maze procedure for studying spatial learning in the rat. J. Neurosci. Methods 11 (1), 47–60.

Mueller, C., Luehrs, M., Baecke, S., Adolf, D., Luetzkendorf, R., Luchtmann, M., et al., 2012. Building virtual reality fMRI paradigms: a framework for presenting immersive virtual environments. J. Neurosci. Methods 209 (2), 290–298.

Muller, E., Bednar, J., Diesmann, M., Gewaltig, M., Hines, M., Muller, E., et al., 2015. Python in neuroscience. Front. Neuroinf. 9, 14–17.

Newman, E.L., Caplan, J.B., Kirschen, M.P., Korolev, I.O., Sekuler, R., Kahana, M.J., 2007. Learning your way around town: how virtual taxicab drivers learn to use both layout and landmark information. Cognition 104 (2), 231–253.

Nishimoto, S., Gallant, J.L., 2011. A three-dimensional spatiotemporal receptive field model explains responses of area mt neurons to naturalistic movies. J. Neurosci. 31 (41), 14551–14564.

Olton, D.S., Samuelson, R.J., 1976. Remembrance of places passed: spatial memory in rats. J. Exp. Psychol. Anim. Behav. Process., 97–116.

Patel, T.P., Man, K., Firestein, B.L., Meaney, D.F., 2015. Automated quantification of neuronal networks and single-cell calcium dynamics using calcium imaging. J. Neurosci. Methods 243, 26–38.

Peirce, J.W., 2007. Psychopy-psychophysics software in Python. J. Neurosci. Methods 162 (1–2), 8–13.

Perkel, J.M., 2015. Programming: pick up Python. Nature 518 (7537), 125–126.

Pitimon, I., Nintanavongsa, P., 2014. An IPv6 network congestion measurement based on network time protocol. TENCON 2014-2014 IEEE Reg 10 Conf., 4–7.

Ravassard, P., Kees, A., Willers, B., Ho, D., Aharoni, D., Cushman, J., et al., 2013. Multisensory control of hippocampal spatiotemporal selectivity. Science 340 (6138), 1342–1346.

Sato, N., Sakata, H., Tanaka, Y., Taira, M., 2004. Navigation in virtual environment by the macaque monkey. Behav. Brain Res. 153 (1), 287–291.
Shen, H., 2014. Interactive notebooks: sharing the code. Nature, 5–6.
Slater, M., Antley, A., Davison, A., Swapp, D., Guger, C., Barker, C., et al., 2006. A virtual reprise of the Stanley Milgram obedience experiments. PLoS One 1 (1).
Slobounov, S.M., Ray, W., Johnson, B., Slobounov, E., Newell, K.M., 2015. Modulation of cortical activity in 2D versus 3D virtual reality environments: an EEG study. Int. J. Psychophysiol. 95 (3), 254–260.
Straw, A.D., 2008. Vision egg: an open-source library for realtime visual stimulus generation. Front. Neuroinform. 2, 4.
Tarr, M.J., Warren, W.H., 2002. Virtual reality in behavioral neuroscience and beyond. Nat. Neurosci. 5 (Suppl), 1089–1092.
Taylor II, R.M., Hudson, T.C., Seeger, A., Weber, H., Juliano, J., Helser, A.T., 2001. VRPN: a device-independent, network-transparent VR peripheral system. Proc. ACM Symposium Virtual Real Software Technol., 55–61.
Topalidou, M., Leblois, A., Boraud, T., Rougier, N.P., 2015. A long journey into reproducible computational neuroscience. Front. Comput. Neurosci. Front. 9, 30.
Wallisch, P., Lusignan, M., Benayoun, M., Baker, T.I., Dickey, A.S., Hatsopoulos, N.G., 2009. Matlab for Neuroscientists. Academic Press.
Washburn, D.A., Astur, R.S., 2003. Exploration of virtual mazes by rhesus monkeys (*Macaca mulatta*). Anim. Cogn. 6 (3), 161–168.
Weidemann, C.T., Mollison, M.V., Kahana, M.J., 2009. Electrophysiological correlates of high-level perception during spatial navigation. Psychon. Bull. Rev. 16 (2), 313–319.
Zheng, F., Whitted, T., Lastra, A., Lincoln, P., State, A., Maimone, A., et al., 2014. Minimizing latency for augmented reality displays: frames considered harmful. IEEE Int Symp Mix Augment Real. 2014, 195–200.