PTVR - a visual perception software in Python to make virtual reality experiments easier to build and more reproducible

Eric Castet, Jérémy Termoz-Masson, Sebastian Vizcay, Johanna Delachambre, Vasiliki Myrodia, Carlos Aguilar, Frédéric Matonti, Pierre Kornprobst

Title

PTVR – a visual perception software in Python to make virtual reality experiments easier to build and more reproducible.

Authors:

Eric Castet[1,], Jérémy Termoz-Masson[2], Sebastian Vizcay[2], Johanna Delachambre[2], Vasiliki Myrodia[1], Carlos Aguilar[3], Frédéric Matonti[4], Pierre Kornprobst[2]

Affiliations:

1 Aix Marseille Univ, CNRS, LPC, Marseille, France

2 Université Côte d'Azur, Inria, France

3 Clubdes3, 211 Promenade des Anglais, Nice, France

4 Centre Monticelli Paradis d'Ophtalmologie, Marseille, France

Funding:

Development of PTVR was supported by grants from the Carnot Cognition Institute, the NeuroMarseille Institute and the ANR (20-CE19-0018).

Abstract

Researchers increasingly use Virtual Reality (VR) to perform behavioral experiments, especially in Vision Science. These experiments are usually programmed directly in so-called game engines that are extremely powerful. However, this process is tricky and time-consuming as it requires solid knowledge of game engines. Consequently, the anticipated prohibitive effort discourages many researchers who want to engage in VR. This paper introduces the Perception Toolbox for Virtual Reality (PTVR) library, allowing visual perception studies in VR to be created using high-level Python script programming. A crucial consequence of using a script is that an experiment can be described by a single, easy-to-read piece of code, thus improving VR studies' transparency, reproducibility, and reusability. We built our library upon a seminal open-source library released in 2018 that we have considerably developed since then. This paper aims to provide a comprehensive overview of the PTVR software for the first time. We introduce the main objects and features of PTVR and some general concepts related to the 3D world. This new library should dramatically reduce the difficulty of programming experiments in VR and elicit a whole new set of visual perception studies with high ecological validity.

Introduction

It has become quite clear, especially in the last decade, that Virtual Reality (VR) is a valid and high-potential technology to build very sophisticated experiments in behavioral vision science and psychological research in general (Cipresso et al., 2018; Gelder et al., 2018; Huygelier et al., 2019; Kourtesis et al., 2019; Parsons, 2015; Rizzo et al., 2021; Scarfe & Glennerster, 2015; C. J. Wilson & Soranzo, 2015). To cite without comprehensiveness a few recent examples related to vision science, experiments have been performed in fields as diverse as binocular vision and stereopsis (Bankó et al., 2022; Levi, 2023), neuropsychological assessment of visual attention (Foerster et al., 2019), visual search (David et al., 2021), visual perception with body/head-movements or with freely moving observers (Bai et al., 2019; Scarfe & Glennerster, 2015), self-location and self-motion perception (Luu et al., 2021; Nakul et al., 2020), visuo-motor control of reaching and pointing movements (Karimpur et al., 2020; Wiesing et al., 2021), testing and training of different sensorimotor functions such as mobility (E. L. Bowman & Liu, 2017) or visuo-motor functions in low vision and ophthalmogogy (Crossland et al., 2019; Soans et al., 2021).

When considering this now large literature, it appears that many of these experiments, especially in recent years, have been programmed directly in one of the two most popular game engines – Unity and Unreal. The Unity engine is more often used (e.g. in nearly all papers cited above) than the Unreal engine (e.g. Hornsey et al., 2020) probably because of Unity's more intuitive interface.

It seems therefore that using game engines has become *de facto* an established way of designing and running valid experiments. There are probably many reasons but it is worth summarizing the main significant and specific advantages of game engines in this context.

First, these game engines are immensely powerful in their ability to model the 3D world and its physical laws and to allow subjects to interact with this world. It is thus possible to construct very sophisticated environments and record multiple parameters of a subject's behavior interacting with this virtual world (e.g. body, head, hand, and more recently eye movements). Second, it is simple with game engines to deal with compatibility issues related notably to the use of VR equipments of different brands. The consequence of this latter point is that experiments can rely on a multiplicity of sophisticated VR equipments (headset, trackers, etc...) whose power and cost have dramatically evolved in opposite directions within a short period. The amplitude and rapidity of this evolution is quite obvious for instance in an important review done less than 10 years ago on the advantages of VR to build experiments for vision science (Scarfe & Glennerster, 2015). Third, game engines can be used for free by scientists as long as large profits do not result from the programmed experiments.

Of course, game engines have their drawbacks. From many informal discussions with colleagues having used VR in their investigations, probably the worst and most cited drawback is the complexity of building an experiment, and as a consequence the slow learning curve of this process. Part of this complexity is not specific to game engines, it is simply due to the general complexity of VR which for instance implies mastering many key principles of 3D geometry. However, another part of this complexity is due, in our opinion, to the general structure and logic of the game engines methods allowing to build an experiment. Many researchers used to build experiments with a script programming language such as PsychoPy (Peirce et al., 2019) find it difficult to use the GUI of game engines. Programming in a game engine means that the programmer's task is to enter the whole content of an experiment in an immense tree of menus and sub-menus. In addition to setting hundreds of parameters (or much more in many experiments) in the GUI, it is often necessary for the programmer to write numerous bits of code (in C# in Unity) that are buried in the labyrinthic tree of the graphical interface. This complexity is exacerbated by the absence of a "main" file which would provide a clear entry point to understand and organize the code.

This general observation concerning the complex structure and logic of game engines will be referred to hereafter as the "game engines complexity" – note that this expression is merely a convenient shortcut.

While game engines are already notorious for inducing a slow learning curve of the programming process, the "game engines complexity" induces additional negative aspects that are points of special concern for researchers. The essence of these problems is that there is no easy and robust way for researchers and engineers to critically analyse and improve the distributed code that was used to create an experiment (L. Aguilar et al., 2022; Grübel, 2023). To understand this point and its importance, it is useful to consider the FAIR for Research Software ("FAIR4RS") principles (Benureau & Rougier, 2018; Chue Hong et al., 2022; Lamprecht et al., 2020) These principles aim at guiding software creators to make their software FAIR – ie. Findable, Accessible, Interoperable and Reusable. As stated clearly in the Introduction of Chue Hong et al. (2022) : "The ultimate goal of FAIR is to increase the transparency, reproducibility, and reusability of research. For this to happen, software needs to be well-described (by metadata), inspectable, documented and appropriately structured so that it can be executed, replicated, built-upon, combined, reinterpreted, reimplemented, and/or used in different settings".

Based on these considerations, it should be clear why "game engines complexity" does induce problems that are specific to research. As already explained, the code in game engines is split and

distributed in many directories and subdirectories, thus requiring a complex cognitive approach to really understand this code. It is thus very difficult, tedious and time-consuming to inspect and comprehend the code created by other researchers, especially if the Methods section of a paper simply states that “the experiment was built directly into the game engine”. The same is true if one wants to execute, check, replicate, …, reimplement, or use the experiment’s code in different settings. In short, experiments built with a game engine do not allow researchers to “increase the transparency, reproducibility, and reusability of research”. These issues are therefore quite far-reaching and have to be considered in the broad framework of the Open Science Framework (Munafò et al., 2017; National Academies of Sciences, Engineering, and Medicine, 2019; Nosek et al., 2015; Open Science Collaboration, 2015; *UNESCO Recommendation on Open Science - UNESCO Digital Library*, n.d.). Put bluntly, an experiment should be described by a clear and structured code that can be made available so that “… code can be reused by others.” (Nosek et al., 2012).

The considerations above led us in 2018 to the following question: would it be possible to keep the advantages of a VR game engine, while being able to make the code of an experiment “usable” in the sense defined above by other researchers? To answer this question, we were inspired by the work of computer programming scientists of the Max Planck Institute for Software Systems (MPI-SWS) who presented their work at the 2018 European Conference on Visual Perception (Mathur et al., 2018). These scientists created an opensource toolbox allowing researchers to write a script in Python to build a VR experiment. Their key idea was that this script was actually used (behind the scene) to build a VR experiment with Unity as a backend. Their toolbox was thus able to contain the whole content of a Unity experiment within a single script. Another asset of this toolbox was that Python was used to write the scripts. The numerous advantages of using Python scripts for scientific research have long been recognised, especially in the vision science community as testified by the success of PsychoPy (Peirce, 2007, 2009), an opensource software often used by researchers to build experiments on 2D monitors.

We therefore decided in 2019 to continue the development of this opensource toolbox called PTVR (Perception Toolbox for Virtual Reality - https://ptvr.inria.fr/). Since then, we have continuously been developing and extending PTVR and part of our work was presented at the 2022 ECVP conference (Castet et al., 2022). This paper aims at providing for the first time a comprehensive overview of the PTVR software.

Methods

Code and Equipment

PTVR is a free and open-source software distributed under the GPL-3.0 license. It is freely available on the PTVR website (https://ptvr.inria.fr/).

At the time of writing, the version of PTVR is 0.11.3 and it is based on Unity version 2020.3.34f1.

PTVR has been tested with HTC Vive Pro (Eye) headsets (and their controllers) connected to VR-ready PCs (as specified by HTC) running under Windows 10. We have also tested eye tracking capabilities with the Sranipal package from Vive (see the section “An experiment and its output files”). We are closely following the development of OpenXR which defines the standard for XR hardware (headsets, controllers and eye-tracking) to make future versions of PTVR work with other headsets.

Every effort was made to improve the readability and consistency of our Python code by following as closely as possible the PEP 8 coding style (https://peps.python.org/pep-0008/).

PTVR architecture

The general PTVR architecture is schematically represented by the flowchart shown in Figure 1. The parts that constitute the toolbox itself, i.e. the parts that have been developed by PTVR developers, are highlighted in light blue (PTVR Python Library, PTVR Unity Library and PTVR.exe).

When creating an experiment, the starting point is to write a PTVR script that describes the experiment: this is done by using features available in the PTVR Python Library. Once a script is ready to run, executing the script performs two successive steps: (1) it creates a JSON file containing all the parameters that will be necessary for Unity to run the experiment, (2) and then, it launches the PTVR.exe application which processes the JSON file, runs the experiment thanks to all the parameters contained in the JSON file and eventually creates the experiment’s output files.

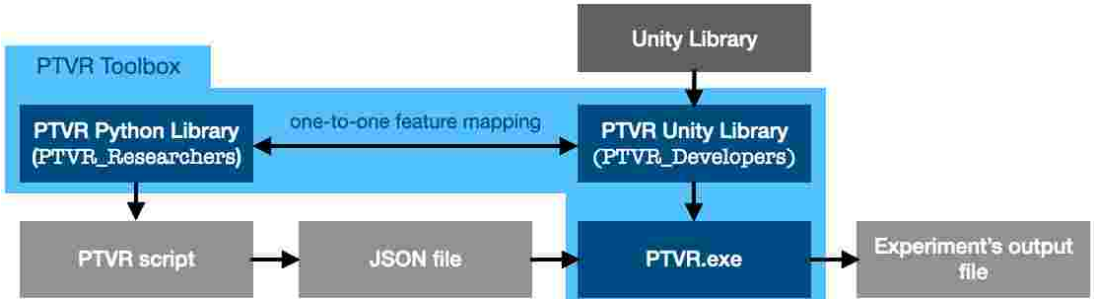

Figure 1 : Overview of the PTVR architecture. The PTVR toolbox (highlighted in light blue) contains a PTVR Python Library allowing users to write their experiments in a PTVR script (in Python). Running this script generates a JSON file which can then be deserialized thanks to the PTVR.exe executable in order to run the experiment and produce an output file. This executable integrates features described in the PTVR Unity Library which itself implements in Unity the features available in the PTVR Python Library.

To allow this flow of operations, the PTVR developers have previously performed the following actions. First, they have created a one-to-one feature mapping between the PTVR Python library and the PTVR Unity library. For instance, as an illustration, they have created simple objects (e.g. spheres, cubes) which are available to users thanks to the PTVR Python library, and in parallel they have created a corresponding object in the PTVR Unity Library. This latter library is written in C# as required by Unity. Second, once the developments above were finished, the developers have created the PTVR.exe application by compiling it based on the PTVR Unity Library and on the standard Unity Library. In sum, the PTVR.exe application is a Unity application that is able to use a JSON file to instantiate all the features that are contained in the PTVR Python library.

It is important to emphasize here that PTVR goes further than simply exposing some of the features available in Unity. PTVR also creates new features that are absent in Unity and at the same time considered useful by vision scientists (e.g. the Tangent Screen object, the perimetric Coordinate System – see the corresponding subsections in the Results Sections).

PTVR Users

PTVR can be used in two different ways depending on the needs of two different broad classes of “users”.

- Our first target user is a person who wants to program a VR experiment, usually a researcher or an engineer (see Figure 2). This user is referred to as a “researcher” for convenience. The first

step for this PTVR user is to write or edit a PTVR script. Note that, in this context, this kind of user is sometimes called a “programmer” and this should not be confused with a PTVR “developer”. This user makes use of the PTVR Python library which contains functions and objects that are written in a high-level language. This high-level language is meant to be easily understood primarily by vision scientists (for instance with the goal of creating an object in a “perimetric” coordinate system). The second step for this PTVR user is to execute the PTVR script, which will launch the experiment and eventually create files describing the outputs of the experiments (subjects’ responses, head and/or gaze movements, …). In sum, this PTVR user will write/edit a Python script, run it and analyse the output files.

Note that, in this scenario, neither Unity nor the PTVR Unity Library need to be installed.

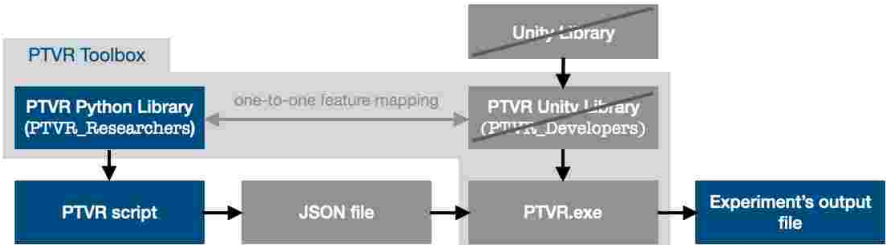

Figure 2 : Flowchart of the steps followed by the first type of PTVR users (“researchers”). These users write a PTVR script describing their experiment by using the PTVR Python Library. When they execute the script, the experiment is launched and it produces the output file. Internally, a JSON file is created and interpreted by the PTVR.exe executable.

- Our second target user is a person running an experiment without any knowledge of PTVR and without knowing about the existence of an underlying PTVR script (see Figure 3). This user is referred to as an “operator” for convenience. This person might be a researcher running an experiment whose script has been written by colleague. This could also be a person running rehabilitation exercises with patients, for instance a vision care professional working with low vision patients. This could also be a teacher running an experiment for educational purposes : for example a 3D geometry teacher or a statistics teacher explaining the 3D principles necessary to visualize multiple regression. In these cases, the PTVR user simply needs to run PTVR.exe and select the JSON file shared by the person who created the experiment.
In some cases, this user might also have the responsibility of processing the data recorded in the experiment’s output file.

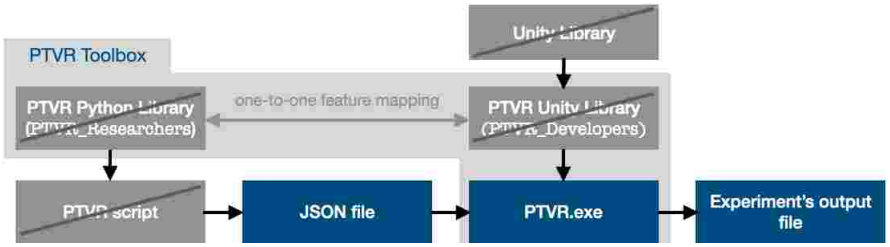

Figure 3 : Flowchart of the steps followed by the second type of PTVR users ("operators"). These users simply need to run PTVR.exe and select the JSON file of the experiment they want to run. In this case, PTVR.exe is the only element they need from the PTVR Toolbox

Version-control system

To work in an open-science framework, it has become essential to provide a version-control system (VCS) to keep track of the successive versions of an open-source code (Blischak et al., 2016; G. Wilson et al., 2014). This is for instance quite clear in a report of the National Academy of Sciences stating that " Scientists are increasingly adopting it as a necessary piece in the **reproducibility** toolbox" (National Academies of Sciences, Engineering, and Medicine, 2019 - p. 114).

There are actually two PTVR repositories to provide a clear organisation of the code:

- the PTVR_Researchers repository

This repository is targeted at both kinds of PTVR users : "researchers" and "operators" (see section "PTVR users"). "Researchers" will find in this repository the elements necessary to program and run an experiment, namely the PTVR Python library and the PTVR.exe application (see Figure 2). "Operators" will only use the PTVR.exe application (see Figure 3).

These users do not need to install Unity on their PC.

- the PTVR_Developers repository

This repository is targeted at PTVR "developers" (see section "PTVR users") and only contains the PTVR Unity library (see Figure 1). The task of "developers" is to improve this library from one release to the other.

These users need to install Unity on their PC.

Results

This section mainly concerns the PTVR "users" who want to write/edit a PTVR script and run the corresponding experiment (see section "PTVR users"). This section thus summarizes the points that will help users to write/edit a PTVR script, and also emphasizes, when relevant, the PTVR features that are especially useful for vision scientists (and absent in Unity).

Learning PTVR with online documentation and demo scripts

Learning how to use a program obviously relies on the availability of a clear documentation containing numerous explanations and tutorials. A good example, in our opinion, is the success of the open source PsychoPy software that owes a lot to its online documentation besides its own merits (Peirce et al., 2019). This is why we made every effort to provide a clear and extensive documentation that is freely available on the PTVR website (https://ptvr.inria.fr/).

First, learning how to write scripts in a program such as PTVR does not only rely on its documentation. It also relies on the availability of "demo scripts" that can highly accelerate the learning curve of this language if they are designed with an intentional didactic purpose. Here again, it is likely that the popularity of the PsychoPy software is largely due to its numerous didactic demo scripts. In PTVR, many didactic demo scripts have therefore been created. Overall, these scripts cover all aspects of the features that are currently available in PTVR (i.e., even some that are not yet covered in the documentation).

Second, based on the observation that it is difficult to explain issues related to 3D geometry with text only (as is often the case in many online documentations), the PTVR documentation presents the following original characteristics:

- Numerous 3D figures help users get a better mental representation of the 3D scenes that are created in the VR headset. This point is so important that these 3D figures are often displayed with an animation to further help visualize the 3D geometry of figures. These figures are mostly designed with the open source software Geogebra (https://www.geogebra.org/), a famous tool used world-wide to teach and learn maths and geometry. PTVR identifies some of these figures as "immersive" because they represent the immersion experienced by users when running the corresponding demo scripts.
- Movies can be watched to showcase some key demo scripts. These movies are generated with film editing techniques whose principle is that the back of an observer is

filmed and superimposed with a movie displaying the content of the headset viewport. For instance, if the observer's head is moving rightward in real life, then the whole VR environment is moving leftward in the movie. Although this kind of presentation is far from simulating what an observer really perceives in the VR headset, it is however helpful to get a quick understanding of the dynamic aspects of some VR scenarios. Some of these movies illustrate the experiments presented in the section "Use cases". All movies are available in the "Media" menu of the PTVR website (https://ptvr.inria.fr/).

Finally, cheatsheets are available in a specific section of the documentation to provide a comprehensive overview of the most important PTVR features (e.g. what are the different ways of specifying the positions of objects?).

Core elements for the creation of virtual interactive 3D worlds

Key role of scenes and objects

Writing a PTVR script first necessitates a clear understanding of the key role of scenes and objects. These two terms have a relatively intuitive meaning from a user's perspective. An object is a visual entity such as a primitive 3D shape (cube, sphere, etc…) or a text in the current PTVR version, and it will represent more complex 2D or 3D visual objects in future PTVR versions. A scene can be considered as a static snapshot that displays a set of PTVR objects for a certain duration.

The scene is a core element because nothing will be displayed in the VR headset if no scene has been created in the PTVR script. If at least one scene has been created, then running the script will successively display the scenes (each with its programmed duration) as illustrated in Figure 4 for a sequence of two scenes (each containing two cones). These two scenes also illustrate that displacing an object from one scene to another can be used to create motion or "apparent motion" (Braddick, 1980). In this case, the most distant cone is moving away.

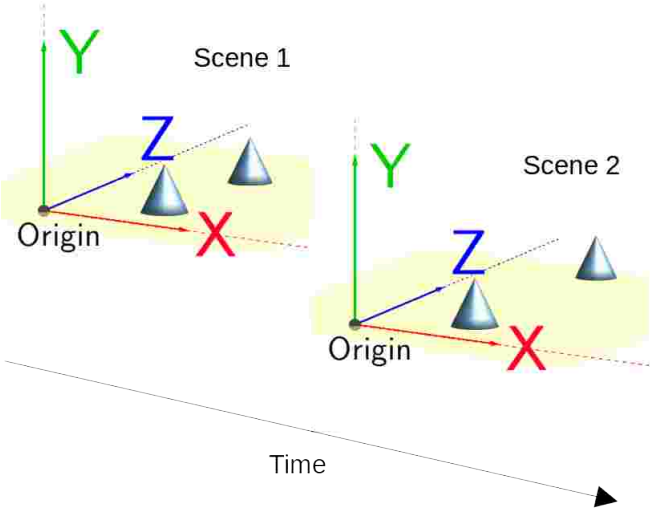

Figure 4 : Running a script in PTVR will display either a single scene or a sequence of "scenes". Here two scenes, each containing two objects, are displayed in a row.

As will be explained below in more concrete details, each scene is able to handle its own interactivity with a high level of sophistication. For instance, the duration of the first scene might depend on some external event like a trigger press while the duration of the second scene might depend on other events such as the direction of the subject's head. Or several of these events might be jointly taken into account for any given scene.

To get a better understanding of how to program scenes in PTVR scripts, the next sub-sections will give some basic explanations about the key principles underlying their logic.

Creating the world and the scenes

Before creating a scene, it is mandatory to create the 3D world that will contain the scenes. This is achieved by the first line of the following code snippet.

```
(code snippet #1)
my_world = PTVR.Visual.The3DWorld ()
my_scene = PTVR.Stimuli.Scenes.VisualScene ()
```

The mandatory call to `PTVR.Visual.The3DWorld ()` is similar in logic to the Psychopy line "win = visual.Window()" which is necessary to create a 2D window on which stimuli will be drawn. Here, the `my_world` object contains information on the 3D world created by PTVR. It is

used for instance when a coordinate transformation is required (see section “Coordinate Transformation”).

The call to `PTVR.Stimuli.Scenes.VisualScene ()` in the second line of code snippet #1 is also mandatory since at least one scene is needed to display objects in the VR headset (see previous section).

Note that a number of code snippets will be used in the present paper to illustrate the simplicity of using PTVR code. For parsimony, we will always assume that these code snippets implicitly contain, as an initial declaration, the two lines of code shown in code snippet #1. It is important to remember this implicit initial declaration of variables to fully understand the code snippets of this paper where some functions/methods will be applied to the `my_world` and `my_scene` objects defined in code snippet #1.

Once the `my_world` and `my_scene` objects have been created, objects can be created. This is illustrated in the following code snippet:

(code snippet #2)

```
my_cylinder = PTVR.Stimuli.Objects.Cylinder ()
my_scene.place (my_cylinder, my_world)
my_world.add_scene (my_scene)
```

A cylinder is first created (first line) and then placed in `my_scene` (second line). Finally, in the third line, `my_scene` is “added” to `my_world,` which is a necessary step to make `my_scene` visible when running the script. Here again, for parsimony, the latter line of code will not be displayed in future code snippets of the paper but it will implicitly included.

Each scene specifies its specific interactivity with events and callbacks

PTVR utilizes an **event**-driven architecture (Van-Roy & Haridi, 2004). Users can define the intended logic of an experiment by creating a list of actions to be performed in response to events that may occur at any time during the program's execution. This event-driven architecture facilitates communication between the custom Python logic, which describes the experiment, and the main PTVR application (PTVR.exe – see section “PTVR architecture”) implemented in Unity using C#

Examples of events utilized in PTVR include user input events, such as key/button presses; timeout events, which are triggered by timers; system events, such as notifications when a scene is loaded, a trial has started, or when a participant has moved beyond a certain point in space.

On the other hand, in PTVR, the specification of actions to be performed when events are triggered is accomplished through **callback** execution. However, Python is not employed in PTVR as a runtime scripting language. Instead, it serves as a language that generates a JSON file describing the experiment. Consequently, researchers are unable to define arbitrary functions as callbacks. To address this, PTVR provides a library of predefined callback objects, which can be utilized as building blocks for complex logic. Since these callback objects are encapsulated as objects, they can be serialized as data into the JSON file.

The PTVR Python callback library empowers researchers to accomplish various common tasks encountered in psychophysics experiments. With this library, researchers can easily perform actions such as displaying and hiding stimuli, modifying properties like color, size, and position of objects, playing audio, presenting text, advancing to the next trial, and recording experimental metrics, among numerous other capabilities.

The connection between events and callbacks in PTVR is established through **interactions**. An interaction serves as a framework that defines which callbacks should be executed when a specific event occurs. These interactions are scoped within a scene, allowing users to create multiple scenes, each listening for different events and executing distinct sets of callbacks. Table 1 provides an illustrative example of how different scenes can be utilized to trigger and execute specific callbacks.

An event might be triggered internally by the PTVR.exe application when this has reached a certain point, like finishing loading some data, or it might be triggered by external sources, such as another application sending a signal or some user interacting with the application by performing some input (key presses, mouse movement, etc). When an event gets triggered, a “callback function” (an event handler function) associated with this given event is executed. For instance, everytime the letter ‘a’ is pressed on the keyboard (a key-pressed event), then letter ‘a’ is displayed on the screen (callback).

In PTVR, an event is something that might happen or not during a scene. As long as a scene is displayed, some EVENTS specific to this scene are listened to by PTVR. A user might for instance

create two scenes, each listening to different sets of events, as schematically shown in Table 1. This table also shows that each event is associated with its own callback.

	Events	Callbacks
Scene 1	When the trigger of the left hand-controller is pressed …	… then change background's color.
	When the duration of the current scene exceeds 200 ms ...	… then end the current scene.
Scene 2	When the "q" key is pressed on the keyboard …	… then end the current scene.

Table 1: Illustration of scene-specific interactivity with an example where each of two successive scenes has a different interactivity.

The desired scene's duration

An important methodological point in behavioural science often concerns the **control of time**. In vision science, it is crucial to control the duration of visual stimuli or the reaction times of subjects in different tasks.

In PTVR, the duration of a visual stimulus is specified by placing this stimulus in a scene (see code snippet #2) and by controlling the duration of this scene. In PTVR, as in unity, controlling a duration relies on the use of events and callbacks (see previous section).

The logic is the following. An event (let's call it "duration_exceeded") is created so that it will be triggered when a certain duration has elapsed from the start of the scene. In addition, a callback is created (let's call it "end_scene") so that it will be executed when the "duration_exceeded" event has been triggered. Quite logically, the "end_scene" callback will contain a piece of code allowing PTVR to stop the display of this scene. This callback will also contain instructions concerning the choice of the next scene to display.

As specifying the duration of a scene is such a common operation when programming experiments, we felt that users might find the use of events and callbacks too time-consuming for this specific operation.

We therefore created a simplified way of creating a scene: instead of creating a "VisualScene" object (as shown in code snippet #3) and creating afterward events and callbacks for this scene (not shown), it is also possible to create a "SimplifiedTimerScene" with the desired duration passed as

an argument. This is illustrated with the example below that will display a text for a “desired” duration of 500 ms:

(code snippet #3)

```
my_scene = PTVR.Stimuli.Scenes.SimplifiedTimerScene (scene_duration_in_ms = 500)
my_text = PTVR.Stimuli.Objects.Text (“Hello”)
my_scene.place (my_text, my_world)
my_world.add_scene (my_scene)
```

Vision scientists know that it is not sufficient to specify a stimulus’ desired duration value in a software to display this stimulus with an actual physical duration corresponding to the desired value. We explain some of the technical details pertaining to this issue in the next section.

The actual scene’s duration

Controlling the actual duration of a stimulus is a key issue in visual psychophysics (“actual duration” meaning “as measured on a monitor with a photo-cell”). Cathode ray tube (CRT) monitors have been used for decades in visual psychophysics for several reasons.

First, a CRT monitor provides actual stimulus durations that correspond to integer multiples of the CRT’s frame duration. For instance, a CRT monitor with a 100 Hz refresh rate (i.e. a frame duration of 10 ms) will provide actual durations such as 50 ms (5 frames), 60 ms (6 frames), etc… and no values inbetween.

Second, with an appropriate software, it is possible to make sure that the actual duration of a stimulus will correspond to a certain number of frames. For instance, a researcher can make sure that the actual stimulus duration will be 50 ms and not 60 ms. This latter achievement is only possible if the stimulus to be displayed does not involve time-consuming processes (e.g. complex calculations necessary to draw this stimulus) that would exceed a certain duration (usually one frame duration). If this duration is exceeded, the actual duration will be longer (usually by one frame duration) than the desired duration.

Since around 2000, many investigations have tested this temporal accuracy issue with different kinds of monitors such as LCD and later OLED monitors (Cooper et al., 2013; Elze, 2010). Temporal accuracy has also been tested with different softwares, operating systems and as a function of different loads imposed by stimulus complexity : a significant and thorough contribution to this investigation, including online internet studies, has been made with the PsychoPy package (Bridges et al., 2020).

Data on temporal accuracy for Virtual Reality headsets are scarcer (Chénéchal & Goldman, 2018; Tachibana & Matsumiya, 2021; Wiesing et al., 2020). They are difficult to interpret as some studies programmed their experiments directly in the Unity or Unreal engines while others used commercial softwares. However it seems that temporal accuracy with VR headsets is not as good as with CRT monitors. In other words, it is difficult to predict in advance whether a given desired duration, even when expressed as a multiple of frame duration, will induce the same actual duration. Therefore, VR cannot be currently recommended for experiments that need an almost perfect control of actual stimulus duration as shown for instance in Bridges et al. (2020). Logically, the same recommendation is currently valid for PTVR.

However, many experiments do not need this almost perfect control of actual duration. The important point for many purposes is rather to be able to measure the difference between the desired and the actual stimulus durations. A first step towards this stringent measure is offered by PTVR. The principle is to record the timestamps associated with the start and with the end of a scene. These two timestamps called “scene_start_timestamp” and “scene_end_timestamp” are stored in the main output file created at the end of a PTVR experiment (see section “An experiment and its output files”). These two timestamps can be used offline to obtain an “estimated duration” of a scene. Most importantly, statistics based on these measures can be calculated to assess the difference between this “estimated duration” and the actual duration. Although it is not our goal in the present work to perform a systematic investigation of this issue as a function of many parameters as in Bridges et al. (2020), we however wanted to present a few preliminary data that turn out to be encouraging. We run a simple demo script that displayed 5000 scenes whose scene durations were set to 100 ms in the script. We chose this desired duration because it is an integer multiple of the frame duration (i.e. 9 frames at the 90 Hz refresh rate of the HTC Vive Pro headset’s screen). Scene “estimated duration” for each of the 5000 scenes was calculated by subtracting “scene_start_timestamp” from “scene_end_timestamp”. The distribution of these scene durations shows that slightly more than 90% of durations are between 100 and 102 ms, and the rest of durations lie between 103 and 112 ms (with two outliers at about 20 ms). These values remain very similar when performing this test several times.

The most important point here is that these measures available in the main output file will allow researchers to compare the “estimated duration” described above with the scenes’ actual duration based on photocell measurements. Researchers who need a very stringent control of scenes duration will then be able to decide whether the statistical uncertainty associated with the scenes’ duration is compatible with their experimental questions.

Coordinate systems (CSs)

Introduction

Working with VR in vision science implies having an accurate control of the positions and orientations of objects in the virtual environment. This actually relies on the use of Coordinate systems (CSs) allowing the researcher to place objects in the virtual world. The stimuli created by researchers usually need more than one CS. For instance, it can be necessary to position objects with respect to the virtual world (virtual-world centered coordinates), with respect to one eye (eye-centered coordinates) or with respect to the subject's head (head-centered coordinates) to name only a few situations.

This is why PTVR developed some useful specific features – that are absent in Unity - for the manipulation of CSs. And, as briefly mentioned above, the `my_world` object created in code snippet #1 has a key role in this manipulation. These features will be described in details in the following sections. To anticipate briefly, PTVR has kept the concepts of global and local CSs that are key concepts in Unity. In addition, PTVR has introduced the concept of "current CS" that is absent in Unity and that allows PTVR to use several CSs within the same script.

Finally, note that the expression "Coordinate System" is synonymous with "Reference Frame" as often encountered in fields such as visual neuroscience or physics. For consistency, PTVR preferentially uses the expression "Coordinate System".

The global coordinate system

By default, the global CS, also referred to as the world CS, is a cartesian CS (aka XYZ system). It is usually represented with the spatial layout shown in Figure 5 (as in Unity), i.e the X and Y axes respectively point rightward and upward while the Z axis recedes in the background. This layout corresponds in mathematics to the so-called left-handed CS. The defaults units are meters. Note that the axes' colors used in the present paper (and in the PTVR documentation) are intentionally the same as in Unity to avoid confusions.

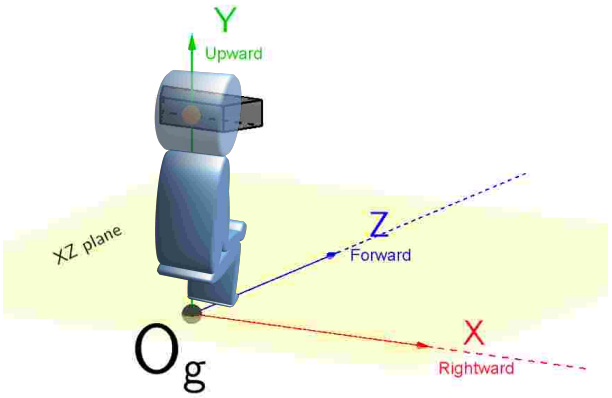

Figure 5 : Canonical representation of the global coordinate system (CS) used by PTVR (and by Unity). Axes' units are in meters. Og stands for "Origin of the global CS". The silhouette represents a subject – here seated - who has performed the calibration of the VR system (see text). Note the headset at the level of his eyes.

It is important to emphasize that the spatial layout of Figure 5 does not give any information on the link between the global CS and the real environment. This relationship is defined by the calibration of the VR system. In other words, once a correct calibration has been performed, the global CS is fixed with respect to the real environment. This fixed relationship is illustrated in Figure 5 thanks to the silhouette representing the subject at the moment when he/she performed a calibration of the VR system. If we assume that the subject's headset direction was aligned with the East direction in the real world at the time of a calibration, then the Z axis of the global CS will always point to the East in the real world for this calibration. Another important pragmatic point illustrated in Figure 5 is that the XZ plane (yellow plane) of the global CS corresponds to the floor of the real world.

Local coordinate systems created with coordinate transformations

In many fields of science, including vision science, it is essential to be able to use several CSs that are called “local” CSs as opposed to the global CS. For instance, in vision science, it is more convenient to define the positions of stimuli with respect to the subject’s cyclopean eye (let’s call it the “cyclopean viewpoint”) rather than with respect to the origin of the global CS. This is achieved by creating a local CS whose origin lies at the cyclopean viewpoint.

Creating a local CS is achieved thanks to a coordinate transformation process which allows to go from one CS to another. This is a key feature of PTVR that proves to be highly convenient in many scientific situations.

In the example above, creating a local CS positioned at the cyclopean viewpoint is easily achieved through coordinate transformation by applying a vertical translation to the global CS so that the origin of the local CS coincides with the viewpoint’s position. After this coordinate transformation, coordinates in this local CS will, as intended, represent the stimuli’s coordinates with respect to the viewpoint. An example of this kind of coordinate transformation is shown in Figure 6.

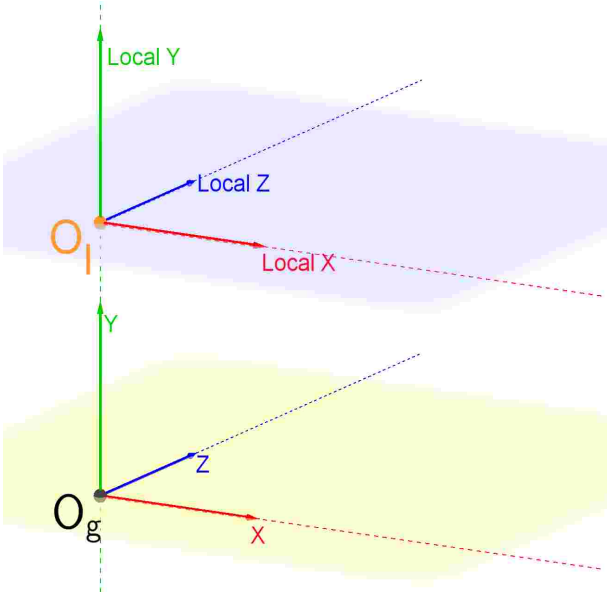

Figure 6 : Example of a coordinate transformation. The local CS in the upper part of the figure is the result of a vertical translation applied to the global CS. Ol stands for “Origin of the local CS”. The blue and yellow surfaces respectively represent the local and the global XZ planes.

In this example, where Ol needs to be positioned at the cyclopean viewpoint, the distance between Og and Ol should be set to the height of the subject's eyes during the experiment. The PTVR code necessary to program this coordinate transformation is presented in the next section.

Using the current coordinate system to place objects

The concept of "current CS" is not present in Unity and is essential in PTVR to place objects in the virtual world. To understand this concept, it is necessary to keep in mind that the different CSs used in a PTVR experiment are created within a script in a certain order. Thus, at any point in a script, there is only one CS that is active in the subsequent script's lines until a coordinate transformation is performed (if any). The CS that is active between two successive coordinate transformations within a PTVR script is called the "current CS".

In other words, at any point within a script, the current CS refers to the last created CS in the script. This CS will be used to place objects in the virtual world until another coordinate transformation occurs (if any).

Let's look at a few examples to see how the concept of current CS is used in PTVR scripts to place objects in the virtual world.

We first want to place a sphere at 1 meter right in front of the global Origin (Og), i.e. at one meter in the Z direction. This is achieved with the following code snippet with occurs at the beginning of a script.

(code snippet #4)

```
# No coordinate transformations have been made yet so that ...
# ... the current CS corresponds to the global CS.
my_sphere = PTVR.Stimuli.Objects.Sphere ( position_in_current_CS = np.array ( [x=0, y=0, z=1] ) )
my_scene.place (my_sphere, my_world)
```

The first uncommented line of code snippet #4 creates a 3D sphere, calls it `my_sphere` and specifies its cartesian coordinates with the triplet (0, 0, 1) which is passed as an argument to the parameter `position_in_current_CS`. The latter name reminds us that the triplet of coordinates passed as an argument refers to the current CS. As code snippet #4 is at the beginning of a script, i.e. when no coordinate transformation has been performed yet, the current CS is the same as the global CS. Consequently, the sphere will be placed at the position shown in Figure 7, i.e. at 1 m in front of Og (if coordinates had not been specified while creating the object, the

position would have been the default (0, 0, 0) triplet). Finally, the last line of code snippet #4 is mandatory to make `my_sphere` visible in the scene named `my_scene`. In other words, once an object is created, it is necessary to assign it explicitly to a scene.

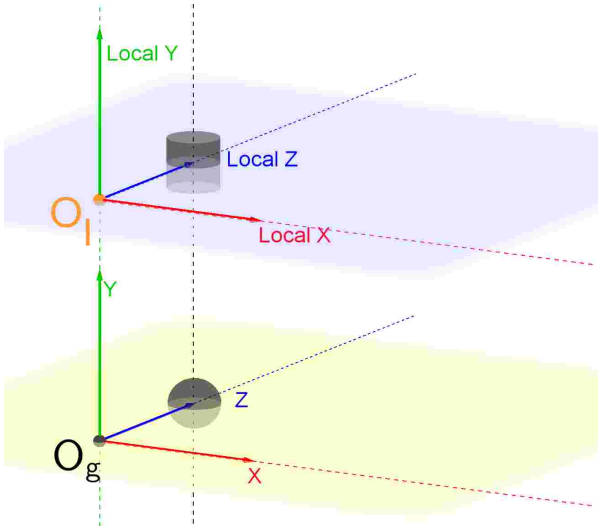

Figure 7 : Illustration of how to use the current CS concept to place objects whose positions are specified in different CSs. The locations of the sphere and of the cylinder are respectively specified in the global and in the local CSs with the same triplet of coordinates (0, 0, 1). The sphere and the cylinder are respectively created by code snippets #4 and #5.

Another interesting example to illustrate how to use the current CS is when the object is created after a coordinate transformation has been performed. Here, we first want to create a local CS resulting from a vertical translation of the global CS. After this coordinate transformation, we want to place a cylinder in front of the origin of this newly created local CS at 1 m (as shown in Figure 7). This is achieved by the following code snippet.

(code snippet #5)

```
my_world.translate_coordinate_system_along_global ( np.array ( [x=0, y=1.4, z=0] )
my_cylinder = PTVR.Stimuli.Objects.Cylinder ( position_in_current_CS = np.array
( [x=0, y=0, z=1] ) )
my_scene.place (my_cylinder, my_world)
```

The vector defining the vertical translation (0, 1.4, 0) in code snippet #5 is specified with global coordinates. This is indicated by the "_along_global" ending in the function's name `translate_coordinate_system_along_global()`. Note that PTVR also provides functions to perform translations specified within the current CS (whose orientation might not be the same as the orientation of the global CS).

The coordinate transformation achieved by code snippet #5 only implies a single translation (it is often called a "shift of origin"). However, two types of coordinate transformations are available in PTVR: translations and rotations (known as "rigid" transformations in mathematics). Code

snippet #6 shows how to perform rotations of the CS. It also illustrates that two coordinate transformations can be combined one after the other with cumulative effects : here two successive rotations of 20° eventually produce a 40° rotation:

(code snippet #6).

```
my_world.rotate_coordinate_system_about_current_x (20)
my_world.rotate_coordinate_system_about_current_x (20)
```

Code snippet #6 is a didactic example as it would have been more concise to achieve a 40° rotation all at once. Note also that, in contrast to code snippet #5, the "_current_x" ending in the function's name `rotate_coordinate_system_about_current_x()` means that the rotations are defined with respect to the X axis of the current CS.

Importantly, all the coordinate transformations available in PTVR are concisely presented in a cheatsheet that can be found in the PTVR documentation (type "cheat" in the documentation's search bar). Cheatsheets summarizing how to place and orient objects are also available in the documentation.

Perimetric coordinates

For any CS defined with cartesian coordinates, it is possible to use other coordinates. As **perimetric** coordinates are of outmost importance in vision science, they have been implemented in PTVR with a user-friendly syntax (this does not exist in Unity). In mathematics, the perimetric CS belongs to the class of spherical CSs.

For simplicity of the figures, the examples in the present section assume that the cartesian CS is the global CS but it could also be any local cartesian CS.

The link between the cartesian CS and the perimetric CS is illustrated in Figure 8. Here, the 3D position of a point P (lying on the yellow sphere) is defined by its 3 perimetric coordinates:

– Half-meridian (from 0° to 360°)

– Eccentricity (from 0° to 180°)

– Radial Distance (in meters): this is the radius (in meters) of the sphere on which P is lying (i.e. distance between the origin of CS and point P).

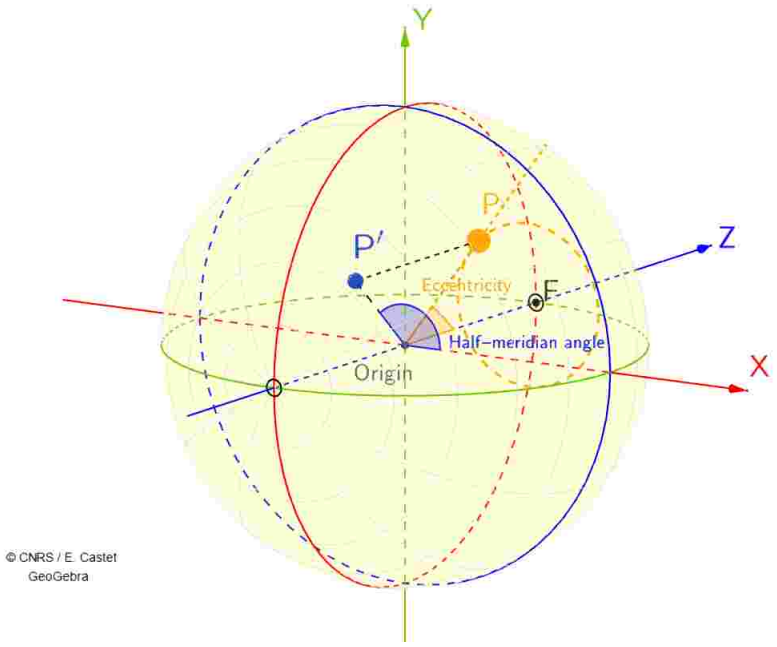

Figure 8 : Example of a perimetric CS superimposed with a cartesian CS. The perimetric coordinates of point P are : half-meridian = 135°, eccentricity = 20°, Radial distance = 1 m.

Figure 9 shows another example to visualize more clearly the link between cartesian coordinates and perimetric coordinates. Here, the cartesian coordinates of P have been specified as (x=0, y=1, z=1) so that its perimetric coordinates are (half-meridian = 90°, eccentricity = 45°, radial distance = √2).

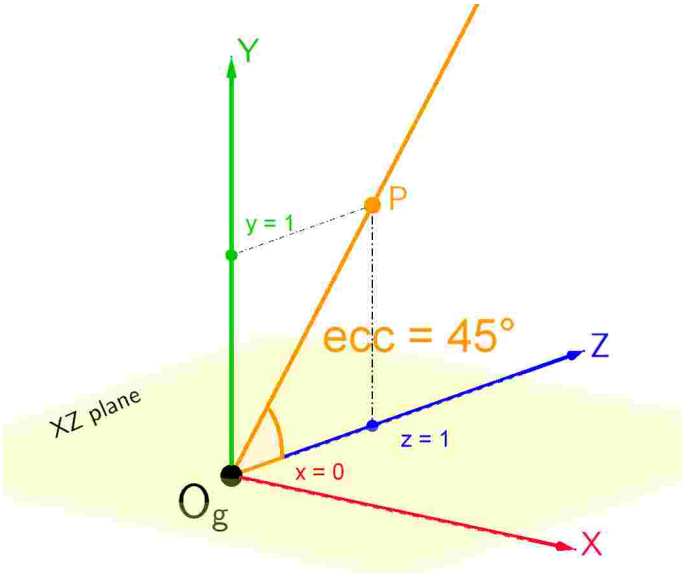

Figure 9 : Simple example to easily visualize the link between cartesian and perimetric coordinates. The cartesian coordinates of point P are (x=0, y=1, z=1) so that its perimetric coordinates are (half-meridian=90°, eccentricity=45°, radial distance = √2 meters) – ecc stands for eccentricity. The radial distance of point P is √(1+1) - i.e., 1.414 - as the Y and Z coordinates of P are 1.

Creating the point P of Figure 9 by specifing its perimetric coordinates is achieved with the following code snippet:

(code snippet #7)

```
# No coordinate transformations have been made yet so that ...
# ... the current CS corresponds to the global CS.
my_point_P = PTVR.Stimuli.Objects.Sphere ()
my_point_P.set_perimetric_coordinates (halfMeridian = 90, eccentricity = 45,
radialDistance = 1.414)
```

In a forthcoming release of PTVR, we plan to add other spherical CSs, notably “azimut-elevation” systems that are ubiquitous in vision science fields related to eye movements and positions, such as binocular processing and eye movement control (Howard & Rogers, 2008).

Main PTVR objects and features

The coordinate system of an object

Every object in PTVR has its own cartesian CS, called the “object’s CS”, which is internally created at the creation of the object. This is illustrated in Figure 10 where a green 2D arrow object is lying in a plane parallel to the XY plane (this arrow object does not exist yet in PTVR and is used here only for convenience as the orientation of an arrow is easy to visualize). The CS of this object is also shown with its origin represented by an orange dot (which corresponds to the object’s position) and with its 3 axes called “object’s X”, “object’s Y” and “object’s Z”.

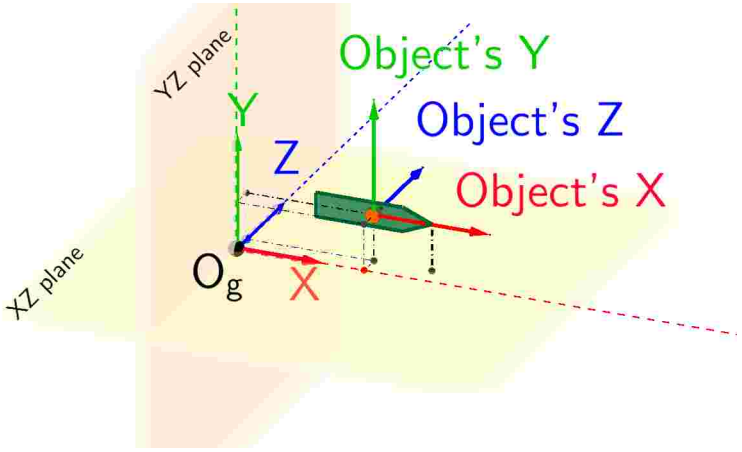

Figure 10 : The CS of an object. Here the object is a 2D green arrow created with its default orientation. The CS of the arrow has its origin at the arrow’s position (orange dot) and its 3 axes are indicated by the “object’s X”, “object’s Y” and “object’s Z” vectors.

Changing the orientation of the arrow object (while keeping its position the same) induces the same orientation change to the arrow object’s CS. For instance, if two successive rotations are applied to the arrow object shown in Figure 10, namely a 40° rotation about the object’s Y axis and a -45° rotation about the ensuing object’s Z axis, this will induce the object’s orientation shown in Figure 11 with a concomitant change of the object’s CS.

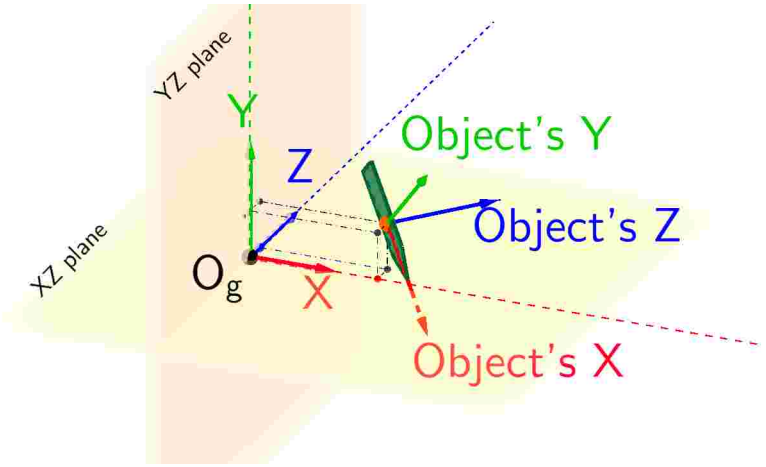

Figure 11 : Orientation of an object's CS as a function of the object's orientation. The green arrow has the same position as in Figure 10 but a different orientation which induces a concomitant rotation of the object's CS (see text).

Although an object's CS is a local CS in the same mathematical way as the local CSs defined in the section "Coordinate systems", its purpose and use are different (these differences are explained in more details in the PTVR documentation). An important role of object's CSs is to provide an easy and quick way to create complex objects made of several sub-parts. For instance, it would be simple to place say a sphere at the pointed tip of the green arrow whatever the arrow's orientation. In other words, it is helpful to use coordinates that are relative to the object irrespectively of the position and orientation of the object. Along these lines, it will be shown in the next sections that using an object's CS is especially useful for some special PTVR objects such as the Flat and Tangent Screens, notably to place text or other objects on these screens.

Thus, although an object's CS is a local CS, we caution users against confusions with the local CSs detailed in the section "Coordinate systems". One important confusion that must not be made concerns the use of the "current CS" concept in a script : it must be remembered that the "current CS" can only refer to the global CS or to a local CS created by coordinate transformations (see code snippets #5 and #6), i.e. it cannot refer to an object's CS.

To reduce the risk of such confusions, we intentionally avoid the use of the "local" term when considering an object's CS either in the text or in the figures of the present work.

Flat and Tangent Screens

The **Flat Screen** (**FS**) is a special PTVR object used to reproduce flat 2D screens that are ubiquitous in real life (e.g. pages of a book, whiteboards in classrooms) and in experimental setups (flat CRT or LCD monitors, …).

An example of FS is shown in Figure 12. In this example, the FS position is (x=0, y=0, z=1) and its orientation is -33 degrees about the X axis (w.r.t. the default vertical orientation) of the FS.

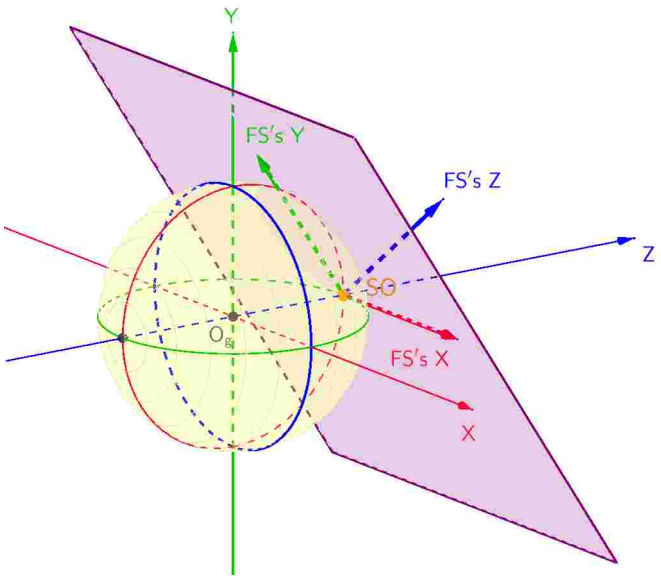

Figure 12 : Example of a Flat Screen (FS). Here the position of the FS is (x=0, y=0, z=1) and its orientation is -33° (see text and code snippet #8). The screen center is called the Screen Origin because it is the origin of the CS of the screen (axes : FS's X, FS's Y, FS's Z).

The FS shown in Figure 12 is created with the following code (this code does not create the yellow sphere):

(code snippet #8)

```
# No coordinate transformations have been made yet so that ...
# ... the current CS corresponds to the global CS.
my_flat_screen = PTVR.Stimuli.Objects.FlatScreen ( position_in_current_CS = np.array ( [x=0, y=0, z=1] )
my_flat_screen.rotate_about_global_x (-33)
```

It is possible to place any visual object (including text) on a FS by specifying the object's cartesian coordinates defined within the CS of the FS (see section "The coordinate system of an object"). These coordinates are the cartesian coordinates specified with respect to the FS's center,

the latter being called the Screen Origin (SO). The axes of this FS's CS are represented in Figure 12 by the vectors FS's X, FS's Y and FS's Z.

Note that a FS can have any rotation. This is in marked contrast to the Tangent Screen (TS) as explained in more details below.

The **Tangent Screen** (TS) is another special PTVR object. It is derived from the FS with several additional features. The two most important of these additional features are:

- additional feature #1: a TS is automatically **tangent** to a notional sphere centered on the origin of the current CS (Figure 13), with the point of tangency defined as the center of the TS (i.e. SO).

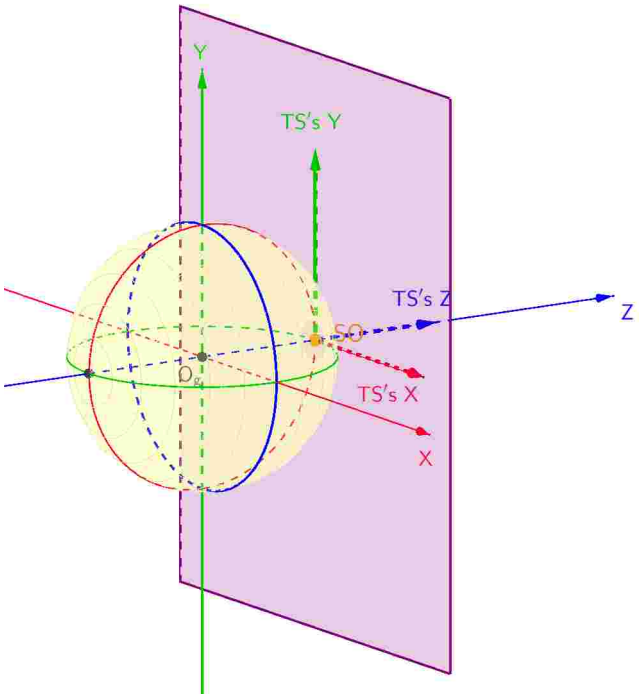

Figure 13 : Example of a Tangent Screen (TS). The TS's position is set by the user to be the same as the FS position in Figure 12 (x=0, y=0, z=1). After the position is set, the TS's orientation is automatically calculated so that it is tangent to *the* yellow sphere at *SO* – see text and code snippet #9.

The TS shown in Figure 13 is created by the following code. Note that the last line of code is commented out.

(code snippet #9)

```
    # No coordinate transformations have been made yet so that ...
    # ... the current CS corresponds to the global CS.
    my_tangent_screen = PTVR.Stimuli.Objects.TangentScreen (position_in_current_CS =
np.array ( [x=0, y=0, z=1] )
    # my_tangent_screen.set_perimetric_coordinates (radialDistance = 0.8, eccentricity =
45, halfMeridian = 180)
```

It is also possible to change the position of a TS right after its creation. This is achieved by uncommenting the last line of code snippet #9. This last line of code now assigns perimetric coordinates to the TS thus leading to a new TS position shown in Figure 14. The new TS position s represented by the light purple segment (note that Figure 14 is a bird's eye view as seen from a notional point lying on the +Y axis of the CS). This TS appears as a segment on this figure as this TS is placed on the horizontal meridian (more precisely, 180° is the leftward horizontal half-meridian). Figure 14 also shows another TS (dark purple) whose eccentricity and half-meridian are respectively 25° and 0°.

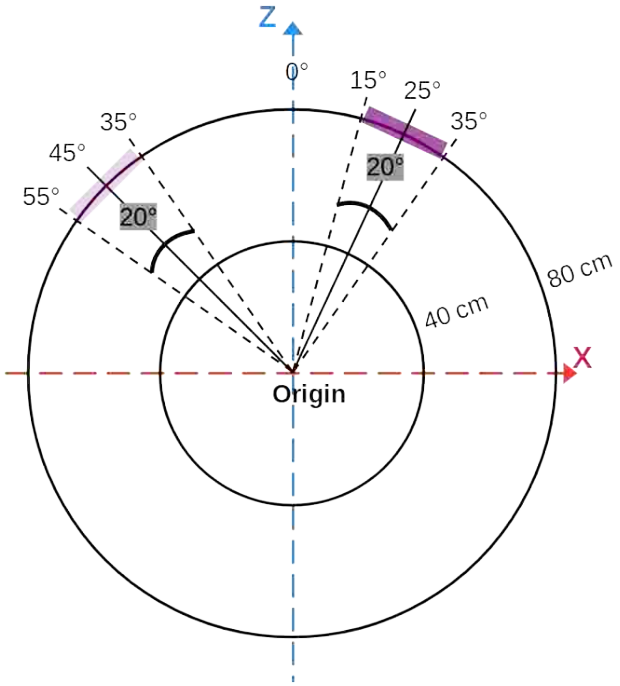

Figure 14 : Two Tangent Screens (TS) in bird's eye view as seen from a notional point lying on the +Y axis of the global CS. The position of a TS is defined by the position of its center. Here the position of the light purple TS is specified in perimetric coordinates by the triplet (eccentricity: 45°, half-meridian:180°, viewing distance: 80 cm), and the position of the dark purple TS is specified in perimetric coordinates by the triplet (eccentricity: 25°, half-meridian:0°, viewing distance: 80 cm). These two TS are tangent to a notional sphere centered on the origin of the current CS (by definition) with a radius of 80 cm.

The main point of code snippet #9 and Figure 14 is to make it clear that simply specifying the position of a TS automatically adjusts its orientation to induce tangency.

- additional feature #2:

A highly convenient feature of TSs, not shared with FSs, is that it is possible to place objects on them thanks to a **perimetric** CS in addition to the cartesian CS already mentioned above for FSs. This 2D perimetric CS is illustrated in Figure 15A. This is a front view as seen from the origin of the current CS.

A **B**

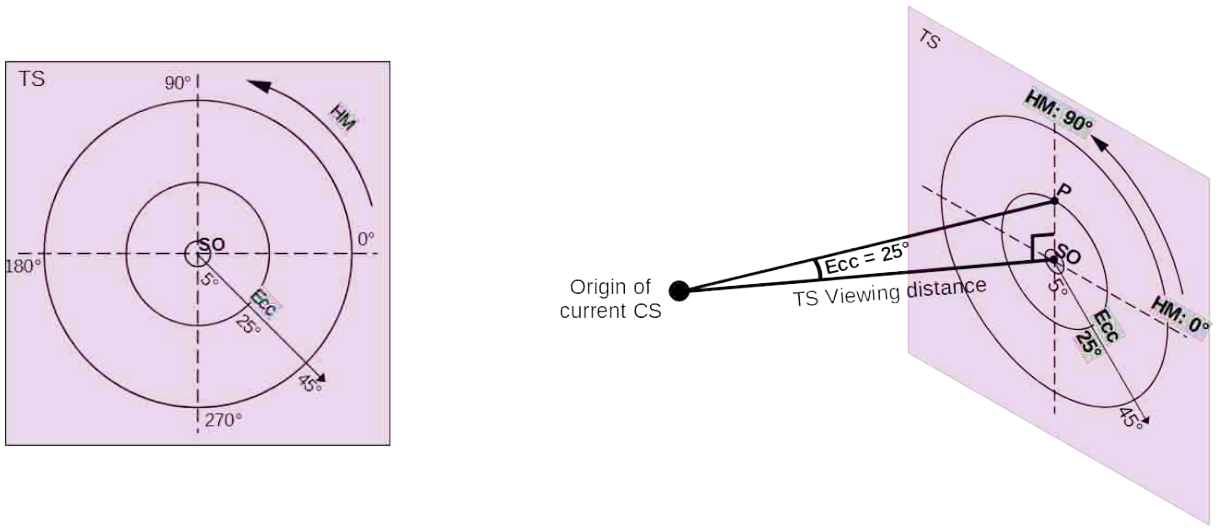

Figure 15 : The perimetric CS used to place objects on a TS. ***(A)*** *the perimetric CS on a TS as seen from the Origin of the current CS. Ecc and HM respectively stand for eccentricity and half-meridian.* ***(B)*** *Example of a small sphere P with a 25° eccentricity on a TS. Eccentricity of any point on a TS is the angle subtended at the Origin of the current CS by the segment from the point to the Screen Origin (S0). Here the eccentricity of sphere P is 25° and its half-meridian is 90°. The TS viewing distance is the distance between the Origin of the current CS and the Screen Origin (SO). Note that the TS, along with the circles lying on it, are represented in (B) with an isometric perspective.*

Placing an object on a TS is achieved by a special PTVR function as shown in the following code snippet.

(code snippet #10)

```
# my_tangent_screen has been created by a previous line.
my_object = PTVR.Stimuli.Objects.Sphere ()
my_object.set_perimetric_coordinates_on_screen (my_tangentscreen = my_tangent_screen, eccentricity=25, half_meridian=90)
```

Code snippet #10 first creates a sphere (called `my_object`) and places this sphere on the TS called `my_tangent_screen` (the sphere is at point P in Figure 15B). This object's position is defined with 2D perimetric coordinates specified within the CS of the TS (see section"the coordinate system of an object"). This specification is a convenient way of placing an object on a TS so that this object will have accurate angular positions (eccentricity and half-meridian) when the subject's cyclopean eye is located at the origin of the current CS (the latter being the CS used to draw `my_tangent_screen`).

Finally, note that all the functions available to place objects on PTVR Screen objects (Flat and Tangent) are detailed in a "position" cheatsheet that can be found in the PTVR documentation (type "cheat" in the documentation's search bar)

Displaying text

The Text object in PTVR has some special features that are especially useful in relation to tests of reading. Its most essential and convenient feature concerns characters' size. It is very important in several fields of vision science to specify the angular size of the characters of a text. It is clearly crucial in fields such as the psychophysics of low vision (Chung et al., 2019; Legge, 2007). Using the angular height of the lower-case "x" letter to specify the angular size of a given font has become a well-established standard in this field as well as in standard tests of reading (Legge & Bigelow, 2011) - see in the section "Use cases" how we adapted the famous MNRead test (Ahn et al., 1995) to VR.

More generally, this is an important constraint in many fields bearing on reading, legibility, ophtalmology, amblyopia, dyslexia, etc… (Goswami, 2015; Levi et al., 2007; Levi & Carney, 2009; Pelli & Tillman, 2008). A very concrete example is the necessity to equate the angular size of different fonts that are experimentally compared in terms of their efficiency – as measured for instance with reading speed (Bernard et al., 2016).

Specifying the angular "x-height" of a text is very easy in PTVR. It is done by passing the "x-height" value (in degrees of visual angle) as an argument when creating a text object as shown in the following code snippet:

(code snippet #11)

```
my_text = PTVR.Stimuli.Objects.Text ( ..., visual_angle_of_centered_x_height = 0.4, ...)
```

This specification has a very accurate meaning: the angular "x-height" (here 0.4 degrees of visual angle) is an angle calculated from the origin of the current CS (i.e., the CS in use just before `my_text` is created),

As an example and for ease of exposition, if we assume that `my_text` is created when the current CS refers to the global CS, the angular value will correspond to an angle calculated from the origin of the global CS (Og). Of course, as for any text displayed on a cardboard or on a 2D monitor in the real world, this specification provides a good approximation of actual angular values only for a text displayed on a Tangent Screen and close to foveal projection.

Incidentally, this "x-height" specification feature turns out to be convenient when creating didactic PTVR scripts where the viewing subject stays grossly at the same position. In this case, specifying say a 1° text will make all labels appear equally legible whatever their positions in the 3D world.

Pointing at a target

As already emphasized, a very useful feature of Virtual Reality for behavioural vision sciences is the possibility of using different kinds of **interactivity** between a subject and the entities contained in the virtual world. In the context of our low vision project, we were particularly interested by the ability of a subject to **select** a target with different kinds of techniques. While the ability of selecting targets with different kinds of controllers has been heavily investigated in the VR literature with normally-sighted persons (Fernandes et al., 2023; Yu et al., 2018), it seems that this topic has been overlooked with low vision persons.

We were especially interested in **pointing** techniques, i.e., the techniques that allow users to select objects that are beyond reach (Bowman & Hodges, 1997; Yu et al., 2018). We have therefore implemented in PTVR two ways of pointing at a target either with the hand controllers or with the head. Some of these developments are very specific to our low vision investigations and will not be presented here. Here, we only present the general PTVR pointing features that might be helpful for the general PTVR user.

Pointing with a hand-controller

The first implemented pointing technique allows a user to point at a target by casting a ray that emanates from the hand-controller (Argelaguet & Andujar, 2013). When this ray intersects an object, selection can be achieved by pressing the trigger of the hand-controller. Head-based ray-casting techniques have also been developed and tested with normally-sighted persons and might be easily implemented in PTVR (Kytö et al., 2018; Qian & Teather, 2017).

Pointing with a head-contingent reticle

The second pointing technique currently implemented is derived from the “aperture” method (Forsberg et al., 1996) and is called the “reticle” technique in PTVR. The general principles of the reticle technique are summarized below.

In the real world, a reticle is a set of lines, often a crosshair, placed into the eyepiece of an optical device such as a telescopic sight or a spotting scope. It is used as a visual aiming aid. The typical reticle used in PTVR is a 2D object as shown in Figure 16A. This kind of pointing is very efficient and useful for instance in visual search tasks (see section “Use cases”).

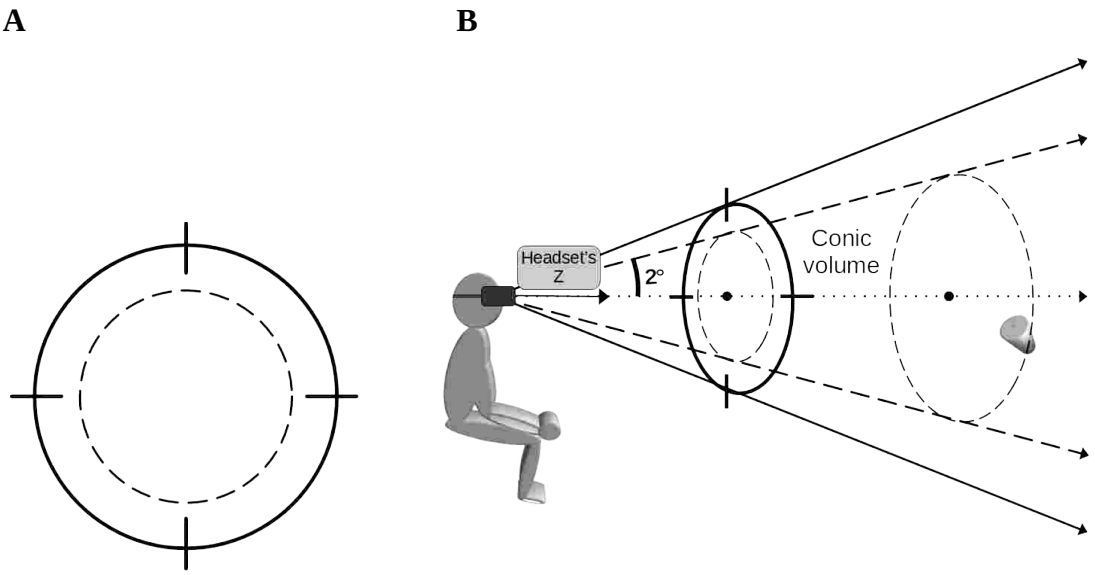

Figure 16 : Definition of the head-contingent reticle. **(A)** *Front view of the 2D reticle : the reticle is a 2D object similar to a crosshair. The inner circle (dashed line) is not visible when the reticle is displayed in the virtual world: it is specified by the angular size of its diameter and defines the activation or acquisition zone of the pointer. When the subject points at a target, the target's center must be within this activation circle to induce valid pointing.* ***(B)*** *Perspective view of the 3D pointing cone. The activation volume corresponding to the activation circle shown in Figure 16A is represented by dashed lines. In this case, the target's center (here a grey cylinder) is within the activation zone : activation is therefore valid. Here the angular diameter of the activation circle is 4°.*

The reticle is always placed on an invisible TS lying on an invisible sphere whose center is the tracked headset. This is illustrated in Figure 17 in a bird's eye view. This Figure also shows that the reticle is head-contingent : the head is straight away in Figure 17A and to the right in Figure 17B. Movements of the head thus allow the user to align the reticle with a distant object.

A valid pointing is illustrated in Figure 16B where the activation conic volume is represented by dashed lines. Here, pointing is considered valid as the target object (here a grey cylinder) has its centre within the activation volume. In this case, some feedback (visual or audio) occurs to warn the user that the target object can now be selected, e.g. by pressing a trigger. The difficulty of the pointing task can be varied by modifying the angular size of the activation circle (see Figure 16A).

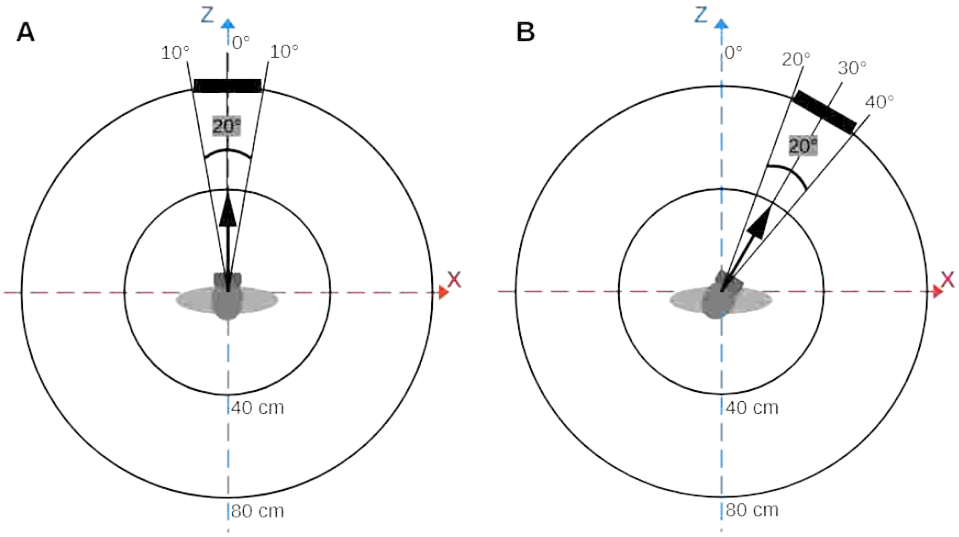

Figure 17 : A reticle in bird's eye view is represented by a black segment for two different head orientations. The reticle is a 2D object (see Figure 16A) always placed on an invisible TS lying on an invisible sphere centered on the headset Note that the activation circle is not shown in this figure. Here, the reticle is placed at a distance of 80 cm from the pointer and has an angular diameter of 20°. **(A)** *The head is pointing straight ahead (i.e., eccentricity is 0°).* **(B)** *The head is pointing in the rightward direction (eccentricity of 30° and half-meridian of 0°).*

Such a head-contingent reticle is currently used in the first author's laboratory to investigate the pointing performance of low vision persons (see section "Use cases").

An experiment and its output files

At the end of a PTVR experiment, i.e. once all the scenes of a script have been displayed, three types of output can be produced to save results. The main output file is systematically produced and contains information allowing researchers to analyse subjects' responses as a function of the trials' parameters. In addition to this main output file, the user can record (with a simple setting in the script) the head data or the eye/gaze data. In the case of eye/gaze data, PTVR uses the open-source SRanipal package from Vive. The output files are event-driven log files (Faison, 2011).

A first example is shown in Figure 18 to emphasize some of the key points of the main output file (more details can be found in the "Files" entry of the PTVR documentation's Index). In this

didactic example, the experiment consists of three trials (each containing one scene). Each scene is displayed until the keyboard space key is pressed. This key press is an event that triggers three main callbacks: 1/ stop the current scene, 2/ display the next scene, and 3/ create a line in the output file to save some relevant variables associated with this event. In this example, all the variables shown in Figure 18 except the last one ("my_variable") are automatically saved as they are considered as essential in all experiments (e.g. "name_of_subject", "scene_id", …). In contrast, a line of code in the script is necessary to save the variables that users have created and that they wish to record. Here the variable "my_variable" in the last column (e.g. here a random number drawn in each trial) is a variable created by the user. It is of course possible to create more complex experiments by adding more events and callbacks in the experiment's script. Each of these events will then create its own line in the main output file.

name_of_subject	trial	scene_id	event_name	event	event_mode	...	callback	...	scene_start_timestamp	scene_end_timestamp	...	my_variable
Martin	1	1	keyboard	space	press	...	EndCurrentScene	...	66417891	66418664	...	7
Martin	2	2	keyboard	space	press	...	EndCurrentScene	...	66418679	66419331	...	3
Martin	3	3	keyboard	space	press	...	EndCurrentScene	...	66419347	66420098	...	6

Figure 18 : Main output file automatically saved after running a script: a few columns are ignored for visual clarity. Each line corresponds to an event (here a space bar key press – see text).

The example shown in Figure 18 also illustrates the key role of the variables called "scene_start_timestamp" and "scene_end_timestamp" (measured in ms). They can be used here to calculate a reaction time (RT). We assume that this fictitious experiment displays a threshold stimulus in each trial and subjects must press the space bar as soon as they detect the stimulus. The RT is thus calculated by subtracting "scene_start_timestamp" from "scene_end_timestamp" as the scene ends when the space bar has been pressed.

Importantly, the role of the "scene_start_timestamp" and "scene_end_timestamp" variables is also crucial to assess statistics concerning the estimated scenes' durations elicited by any ptvr script (see section "The actual scene's duration").

In the future, PTVR will allow the creation of more output files that will perform some higher-level processing of the data already stored in the output files presented above.

Use cases

To illustrate the PTVR capability with more concrete examples, we present in this section three experiments that have already been developed, two of which are directly related to our research in the field of low-vision. These examples show how the features presented in this paper can be used to build complex or famous experiments. The interested reader will find some associated videos and PTVR scripts on PTVR website.

Visual Search

This example script illustrates the importance of PTVR pointing techniques. Visual search is a very well-known research topic. It has been extensively investigated in 2D and the number of investigations of this topic in more realistic 3D environments is growing (Beitner et al., 2021). The visual search PTVR script illustrates how simple it is to create such an experiment. It also illustrates the benefits brought about by PTVR pointing techniques to help subjects give their responses in paradigms for which using the keyboard or buttons is not optimal. We revisited the 3D visual search PTVR script introduced by (Mathur et al., 2018). In this experiment, the subject must find and select a target (here a red cube) among distractors (red and green spheres as well as green cubes) placed around the subject in perimetric coordinates. The user is free to look around and can use the hand controller equipped with a “laser pointer” to point at the target and press the trigger when the target is pointed at. Note that no formal study has been conducted so far with this experiment.

Pointing task with a head-contingent reticle

This example script again illustrates the importance of PTVR pointing techniques but in a more complex experimental design. This use case aims at investigating how well low-vision patients perform a pointing task compared to normally-sighted controls. Our overall motivation behind this specific task is to develop innovative visual aids and methods of low vision rehabilitation (C. Aguilar & Castet, 2017; Calabrèse, Aguilar, et al., 2018; Legge & Chung, 2016), all based on the assets of VR (work supported by the ANR DEVISE project: https://anr.fr/Project-ANR-20-CE19-0018). In this experiment, low vision persons have to move their head to point with a head-contingent reticle at a static target dot in the virtual environment, while the difficulty of the task is varied by controlling the activation size of the pointing reticle (see Figure 16). This is an ongoing study which has led to some preliminary results (Myrodia et al., 2023).

The MNREAD test

This example script illustrates the PTVR capability to display text at an accurately controlled angular size thanks to the well-known x-height measure (Legge & Bigelow, 2011). Among extant reading tests, the MNREAD test (Mansfield et al., 1993) is probably one of the most widely used standardized reading tests whether in clinical or in research contexts. While the well-established standard test is a printed version, a tablet version (iPad) has recently been developed and validated to facilitate the distribution and use of the test (Calabrèse, To, et al., 2018). The example script is a VR version using the original MNREAD sentences in English. The raw data of reading performance (reading speed and errors for each sentence) is displayed in an operator interface and saved in an output file, thus allowing further analysis to extract several indicators of reading performance (Baskaran et al., 2019). Performing the MNREAD test in VR has four main advantages: (1) This technology allows us to display huge angular print sizes, without being limited by a physical screen's size thanks to the 360-degree immersion. (2) It enables us to have absolute control over the experimental conditions, such as the luminance of the real environment. (3) It allows the operator to have a precise control of the distance between the patient's head and the text, which is an important piece of information to monitor and check during a reading test. (4) One can envision new behavioral studies thanks to eye-tracking recording during the test (Calabrèse et al., 2016). Our hope is that this key reading test will be easily used anywhere by many researchers in reading. In sum, easily performing this standardized test in a well-controlled VR environment should help produce a large amount of reproducible data from multiple sites.

Discussion

PTVR (Perception Toolbox for Virtual Reality) is a free and open-source software allowing users to easily build a VR experiment (based on Unity as a backend). The user writes a Python script using PTVR functions and objects whose explicit names make programming intuitive. Then, launching the script will run the experiment (with Unity running behind the scene) and eventually save relevant results (note that the user does not need any Unity knowledge and that Unity is not even installed on the user's PC).

This scripting philosophy can overcome several problems induced by the "game engines complexity" (see section "Introduction"). In short, a PTVR experiment can be quickly and integrally reproduced by anyone possessing the required equipments (which have become relatively cheap in the last years). Beyond this huge asset, a PTVR script also makes an experiment's code more transparent and reusable when compared with the game engine's programming philosophy. This in line with the FAIR for Research Software ("FAIR4RS") principles (Benureau & Rougier, 2018; Chue Hong et al., 2022; Lamprecht et al., 2020). These principles are of outmost importance to improve reproducibility of science (Munafò et al., 2017), but also when researchers want to interact with each other to accurately understand the methods of a VR experiment, or when they interact with engineers to improve an experiment's code. A single script is also a great didactic tool to teach students about the principles of building an experiment by giving them an optimal hands-on experience. All these advantages of a script are especially clear as PTVR uses a high-level language whose syntax and terminology are meant to be easily understood and memorized.

We know of two recent open-source developments that allow researchers to write scripts to build a VR experiment. The first one is integrated within the Psychophysics Toolbox version 3 – PTB-3 (Brainard, 1997). The second one is a set of Python extension libraries for interacting with eXtended Reality displays (PsychXR - http://psychxr.org/). PsychXR is used for HMD support by the PsychoPy package (Peirce et al., 2019). It seems that the main difference between these two developments and PTVR relies on their respective backends. PTVR uses the Unity game engine as a backend (without the user noticing it), whereas these two packages are developed at a lower programming level (using OpenGL for instance). These two different development strategies will have their respective advantages and drawbacks. On the one hand, using Unity as a backend implies that very powerful VR and XR features are already developed in this engine and can thus be relatively quickly "incorporated" into PTVR. In contrast, it will be much more complex to develop

those same features “from scratch” in PTB-3 and in PsychXR. On the other hand, the lower-level developments (based on OpenGL) of PTB-3 and PsychXR might provide more control on low-level aspects of the stimuli (e.g. temporal accuracy), whereas PTVR will be dependent on the Unity code which might not have the same level of control. Apart from these obvious comparisons, we feel that PTVR and these two developments (PTB-3 and PsychXR) are too recent to allow us to make an interesting and well-grounded comparison between their characteristics. More importantly, we believe that future research might show that PTVR and these two packages have their own respective merits and will be used by different categories of users having complementary purposes.

The present work is focused on the methodological aspects of PTVR. Therefore, the main objects and features of PTVR, as well as the benefits they potentially bring about for vision science, have been presented in some details. Other important objects and features of PTVR have not been presented here but they can be discovered in the documentation of the PTVR website, or in the demo scripts provided within PTVR. For instance, it is possible to choose between binocular and monocular viewing, to create 2D objects (called “sprites” in Unity), or to play audio files.

In addition, some key PTVR features are still in development. The priority development aims at helping users build experimental designs based on standard psychophysical procedures ranging from standard ones (e.g. method of constant stimuli) to more sophisticated ones such as adaptive procedures (e.g. staircases). The second priority development aims at allowing users to easily import realistic 3D objects (e.g. animals, tables, fruits) and scenes (e.g. a room) into PTVR. Another set of features to be developed concerns the use of the specifics CSs (e.g. the Harms CS) used in the fields of binocular vision and binocular eye movements (Howard & Rogers, 2008). These CSs are usually difficult to understand and manipulate when they are taught with 2D visual material, which is why some researchers designed different kinds of “ophthalmotropes” (Schreiber & Schor, 2007). These novel PTVR tools, including some form of “ophthalmotropes”, might therefore help teach these notoriously complex CSs to students. In this case, PTVR “experiments” might actually be used as pedagogical aids (Schloss et al., 2021).

To sum up, we hope that PTVR will be a useful toolbox allowing vision scientists and experimental psychologists to easily build VR experiments while sharing/testing/improving the associated transparent codes. These PTVR “experiments” might correspond to real experiments performed by scientists or to pedagogical aids run by lecturers.

Acknowledgments

We thank Hui-Yin Wu for helpful discussions on methodological issues.

References

Aguilar, C., & Castet, E. (2017). Evaluation of a gaze-controlled vision enhancement system for reading in visually impaired people. *PLOS ONE*, *12*(4), e0174910. https://doi.org/10.1371/journal.pone.0174910

Aguilar, L., Gath-Morad, M., Grübel, J., Ermatinger, J., Zhao, H., Wehrli, S., Sumner, R. W., Zhang, C., Helbing, D., & Hölscher, C. (2022). *Experiments as Code: A Concept for Reproducible, Auditable, Debuggable, Reusable, & Scalable Experiments*. https://doi.org/10.48550/ARXIV.2202.12050

Ahn, S. J., Legge, G. E., & Luebker, A. (1995). Printed cards for measuring low-vision reading speed. *Vision Res*, *35*, 1939–1944.

Argelaguet, F., & Andujar, C. (2013). A survey of 3D object selection techniques for virtual environments. *Computers & Graphics*, *37*(3), 121–136. https://doi.org/10.1016/j.cag.2012.12.003

Bai, J., Bao, M., Zhang, T., & Jiang, Y. (2019). A virtual reality approach identifies flexible inhibition of motion aftereffects induced by head rotation. *Behavior Research Methods*, *51*(1), 96–107. https://doi.org/10.3758/s13428-018-1116-6

Bankó, É. M., Barboni, M. T. S., Markó, K., Körtvélyes, J., Németh, J., Nagy, Z. Zs., & Vidnyánszky, Z. (2022). Fixation instability, astigmatism, and lack of stereopsis as factors impeding recovery of binocular balance in amblyopia following binocular therapy. *Scientific Reports*, *12*(1), 10311. https://doi.org/10.1038/s41598-022-13947-y

Baskaran, K., Macedo, A. F., He, Y., Hernandez-Moreno, L., Queirós, T., Mansfield, J. S., & Calabrèse, A. (2019). Scoring reading parameters: An inter-rater reliability study using the MNREAD chart. *PLOS ONE*, *14*(6), e0216775. https://doi.org/10.1371/journal.pone.0216775

Beitner, J., Helbing, J., Draschkow, D., & Võ, M. L.-H. (2021). Get Your Guidance Going: Investigating the Activation of Spatial Priors for Efficient Search in Virtual Reality. *Brain Sciences*, *11*(1), 44. https://doi.org/10.3390/brainsci11010044

Benureau, F. C. Y., & Rougier, N. P. (2018). Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *Frontiers in Neuroinformatics*, *11*, 69. https://doi.org/10.3389/fninf.2017.00069

Bernard, J.-B., Aguilar, C., & Castet, E. (2016). A New Font, Specifically Designed for Peripheral Vision, Improves Peripheral Letter and Word Recognition, but Not Eye-Mediated Reading Performance. *PLOS ONE*, *11*(4), e0152506. https://doi.org/10.1371/journal.pone.0152506

Blischak, J. D., Davenport, E. R., & Wilson, G. (2016). A Quick Introduction to Version Control with Git and GitHub. *PLOS Computational Biology*, *12*(1), e1004668. https://doi.org/10.1371/journal.pcbi.1004668

Bowman, D. A., & Hodges, L. F. (1997). An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, 35–38. https://doi.org/10.1145/253284.253301

Bowman, E. L., & Liu, L. (2017). Individuals with severely impaired vision can learn useful orientation and mobility skills in virtual streets and can use them to improve real street safety. *PLOS ONE*, *12*(4), e0176534. https://doi.org/10.1371/journal.pone.0176534

Braddick, O. J. (1980). Low-level and high-level processes in apparent motion. *Philosophical Transactions of the Royal Society of London B*, *290*, 137–151.

Brainard, D. H. (1997). The Psychophysics Toolbox. *Spatial Vision*, *10*(4), 433–436. https://doi.org/10.1163/156856897X00357

Bridges, D., Pitiot, A., MacAskill, M. R., & Peirce, J. W. (2020). The timing mega-study: Comparing a range of experiment generators, both lab-based and online. *PeerJ*, *8*, e9414. https://doi.org/10.7717/peerj.9414

Calabrèse, A., Aguilar, C., Faure, G., Matonti, F., Hoffart, L., & Castet, E. (2018). A Vision Enhancement System to Improve Face Recognition with Central Vision Loss. *Optometry and Vision Science*, *95*(9), 738–746. https://doi.org/10.1097/OPX.0000000000001263

Calabrèse, A., Bernard, J.-B., Faure, G., Hoffart, L., & Castet, E. (2016). Clustering of Eye Fixations: A New Oculomotor Determinant of Reading Speed in Maculopathy. *Investigative Ophthalmology & Visual Science*, *57*(7), 3192–3202. https://doi.org/10.1167/iovs.16-19318

Calabrèse, A., To, L., He, Y., Berkholtz, E., Rafian, P., & Legge, G. E. (2018). Comparing performance on the MNREAD iPad application with the MNREAD acuity chart. *Journal of Vision*, *18*(1), 8–8. https://doi.org/10.1167/18.1.8

Castet, E., Termoz-Masson, J., Delachambre, J., Hugon, C., Wu, H.-Y., & Kornprobst, P. (2022). PTVR : a user-friendly open-source script programming package to create Virtual Reality experiments. *Perception*, *51*, Oral paper presented at the 44th European Conference on Visual Perception (ECVP) 2022, Nijmegen, The Netherlands. https://doi.org/10.1177/03010066221141167

Chénéchal, M. L., & Goldman, J. C. (2018). HTC Vive Pro Time Performance Benchmark for Scientific Research. *ICAT-EGVE 2018 - International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments*, 4 pages. https://doi.org/10.2312/EGVE.20181318

Chue Hong, N. P., Katz, D. S., Barker, M., Lamprecht, A.-L., Martinez, C., Psomopoulos, F. E., Harrow, J., Castro, L. J., Gruenpeter, M., Martinez, P. A., Honeyman, T., Struck, A., Lee, A., Loewe, A., van Werkhoven, B., Jones, C., Garijo, D., Plomp, E., Genova, F., … WG, R. F. (2022). *FAIR Principles for Research Software (FAIR4RS Principles)* (1.0). Zenodo. https://doi.org/10.15497/RDA00068

Chung, S. T. L., Legge, G. E., Pelli, D. G., & Yu, C. (2019). Visual factors in reading. *Vision Research*, *161*, 60–62. https://doi.org/10.1016/j.visres.2019.06.002

Cipresso, P., Giglioli, I. A. C., Raya, M. A., & Riva, G. (2018). The Past, Present, and Future of Virtual and Augmented Reality Research: A Network and Cluster Analysis of the Literature. *Frontiers in Psychology*, *9*. https://www.frontiersin.org/articles/10.3389/fpsyg.2018.02086

Cooper, E. A., Jiang, H., Vildavski, V., Farrell, J. E., & Norcia, A. M. (2013). Assessment of OLED displays for vision research. *Journal of Vision*, *13*(12), 16. https://doi.org/10.1167/13.12.16

Crossland, M. D., Starke, S. D., Imielski, P., Wolffsohn, J. S., & Webster, A. R. (2019). Benefit of an electronic head-mounted low vision aid. *Ophthalmic and Physiological Optics*, *39*(6), 422–431. https://doi.org/10.1111/opo.12646

David, E. J., Beitner, J., & Võ, M. L.-H. (2021). The importance of peripheral vision when searching 3D real-world scenes: A gaze-contingent study in virtual reality. *Journal of Vision*, *21*(7), 3. https://doi.org/10.1167/jov.21.7.3

Elze, T. (2010). Misspecifications of Stimulus Presentation Durations in Experimental Psychology: A Systematic Review of the Psychophysics Literature. *PLoS ONE*, *5*(9), e12792. https://doi.org/10.1371/journal.pone.0012792

Faison, E. W. (2011). *Event-based programming: Taking events to the limit*. Apress.

Fernandes, A. S., Murdison, T. S., & Proulx, M. J. (2023). Leveling the Playing Field: A Comparative Reevaluation of Unmodified Eye Tracking as an Input and Interaction Modality for VR. *IEEE Transactions on Visualization and Computer Graphics*, 1–11. https://doi.org/10.1109/TVCG.2023.3247058

Foerster, R. M., Poth, C. H., Behler, C., Botsch, M., & Schneider, W. X. (2019). Neuropsychological assessment of visual selective attention and processing capacity with head-mounted displays. *Neuropsychology*, *33*(3), 309–318. https://doi.org/10.1037/neu0000517

Forsberg, A., Herndon, K., & Zeleznik, R. (1996). Aperture based selection for immersive virtual environments. *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, 95–96. https://doi.org/10.1145/237091.237105

Gelder, B. de, Kätsyri, J., & Borst, A. W. de. (2018). Virtual reality and the new psychophysics. *British Journal of Psychology*, *109*(3), 421–426. https://doi.org/10.1111/bjop.12308

Goswami, U. (2015). Sensory theories of developmental dyslexia: Three challenges for research. *Nature Reviews Neuroscience*, *16*(1), 43–54. https://doi.org/10.1038/nrn3836

Grübel, J. (2023). The design, experiment, analyse, and reproduce principle for experimentation in virtual reality. *Frontiers in Virtual Reality*, *4*, 1069423. https://doi.org/10.3389/frvir.2023.1069423

Hornsey, R. L., Hibbard, P. B., & Scarfe, P. (2020). Size and shape constancy in consumer virtual reality. *Behavior Research Methods*, *52*(4), 1587–1598. https://doi.org/10.3758/s13428-019-01336-9

Howard, I. P., & Rogers, B. J. (2008). *Seeing in Depth* (1–2). Oxford University Press. https://doi.org/10.1093/acprof:oso/9780195367607.001.0001

Huygelier, H., Schraepen, B., van Ee, R., Vanden Abeele, V., & Gillebert, C. R. (2019). Acceptance of immersive head-mounted virtual reality in older adults. *Scientific Reports*, *9*(1), 4519. https://doi.org/10.1038/s41598-019-41200-6

Karimpur, H., Eftekharifar, S., Troje, N. F., & Fiehler, K. (2020). Spatial coding for memory-guided reaching in visual and pictorial spaces. *Journal of Vision*, *20*(4), 1. https://doi.org/10.1167/jov.20.4.1

Kourtesis, P., Collina, S., Doumas, L. A. A., & MacPherson, S. E. (2019). Technological Competence Is a Pre-condition for Effective Implementation of Virtual Reality Head Mounted Displays in Human Neuroscience: A Technological Review and Meta-Analysis. *Frontiers in Human Neuroscience*, *13*, 342. https://doi.org/10.3389/fnhum.2019.00342

Kytö, M., Ens, B., Piumsomboon, T., Lee, G. A., & Billinghurst, M. (2018). Pinpointing: Precise Head- and Eye-Based Target Selection for Augmented Reality. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, 1–14. https://doi.org/10.1145/3173574.3173655

Lamprecht, A.-L., Garcia, L., Kuzak, M., Martinez, C., Arcila, R., Martin Del Pico, E., Dominguez Del Angel, V., van de Sandt, S., Ison, J., Martinez, P. A., McQuilton, P., Valencia, A., Harrow, J., Psomopoulos, F., Gelpi, J. L., Chue Hong, N., Goble, C., & Capella-Gutierrez, S. (2020). Towards FAIR principles for research software. *Data Science*, *3*(1), 37–59. https://doi.org/10.3233/DS-190026

Legge, G. E. (2007). *Psychophysics of Reading in Normal and Low Vision*. Lawrence Erlbaum Associates.

Legge, G. E., & Bigelow, C. A. (2011). Does print size matter for reading? A review of findings from vision science and typography. *Journal of Vision*, *11*(5), 1–22. https://doi.org/10.1167/11.5.8

Legge, G. E., & Chung, S. T. L. (2016). Low Vision and Plasticity: Implications for Rehabilitation. *Annual Review of Vision Science*, *2*(1), 321–343. https://doi.org/10.1146/annurev-vision-111815-114344

Levi, D. M. (2023). Applications and implications for extended reality to improve binocular vision and stereopsis. *Journal of Vision*, *23*(1), 14. https://doi.org/10.1167/jov.23.1.14

Levi, D. M., & Carney, T. (2009). Crowding in peripheral vision: Why bigger is better. *Current Biology : CB*, *19*(23), 1988–1993. https://doi.org/10.1016/j.cub.2009.09.056

Levi, D. M., Song, S., & Pelli, D. G. (2007). Amblyopic reading is crowded. *J Vis*, *7*, 1–17.

Luu, W., Zangerl, B., Kalloniatis, M., Palmisano, S., & Kim, J. (2021). Vision Impairment Provides New Insight Into Self-Motion Perception. *Investigative Opthalmology & Visual Science*, *62*(2), 4. https://doi.org/10.1167/iovs.62.2.4

Mansfield, J. S., Ahn, S. J., Legge, G. E., & Luebker, A. (1993). A New Reading-Acuity Chart for Normal and Low Vision. *Noninvasive Assessment of the Visual System (1993), Paper NSuD.3*, NSuD.3. https://doi.org/10.1364/NAVS.1993.NSuD.3

Mathur, A. S., Majumdar, R., & Ghose, T. (2018). Psychophysics Toolbox for Virtual Reality. *Perception*, *48*, Poster presented at the European Conference on Visual Perception (ECVP) 2018, Trieste, Italy. https://doi.org/10.1177/0301006618824879

Munafò, M. R., Nosek, B. A., Bishop, D. V. M., Button, K. S., Chambers, C. D., Percie du Sert, N., Simonsohn, U., Wagenmakers, E.-J., Ware, J. J., & Ioannidis, J. P. A. (2017). A manifesto for reproducible science. *Nature Human Behaviour*, *1*(1), 0021. https://doi.org/10.1038/s41562-016-0021

Myrodia, V., Calabrèse, A., Denis-Noël, A., Matonti, F., Kornprobst, P., & Castet, E. (2023). Pointing at static targets in a virtual reality environment: Performance of visually impaired vs. normally-sighted persons. *Journal of Vision*, *23*(9), 5543. https://doi.org/10.1167/jov.23.9.5543

Nakul, E., Orlando-Dessaints, N., Lenggenhager, B., & Lopez, C. (2020). Measuring perceived self-location in virtual reality. *Scientific Reports*, *10*(1), 6802. https://doi.org/10.1038/s41598-020-63643-y

National Academies of Sciences, Engineering, and Medicine. (2019). *Reproducibility and Replicability in Science*. Washington, DC : The National Academies Press. https://doi.org/10.17226/25303

Nosek, B. A., Alter, G., Banks, G. C., Borsboom, D., Bowman, S. D., Breckler, S. J., Buck, S., Chambers, C. D., Chin, G., Christensen, G., Contestabile, M., Dafoe, A., Eich, E., Freese, J., Glennerster, R., Goroff, D., Green, D. P., Hesse, B., Humphreys, M., … Yarkoni, T. (2015). Promoting an open research culture. *Science*, *348*(6242), 1422–1425. https://doi.org/10.1126/science.aab2374

Nosek, B. A., Spies, J. R., & Motyl, M. (2012). Scientific Utopia II. Restructuring Incentives and Practices to Promote Truth Over Publishability. *Perspectives on Psychological Science*, *7*(6), 615–631. https://doi.org/10.1177/1745691612459058

Open Science Collaboration. (2015). Estimating the reproducibility of psychological science. *Science*, *349*(6251), aac4716–aac4716. https://doi.org/10.1126/science.aac4716

Parsons, T. D. (2015). Virtual Reality for Enhanced Ecological Validity and Experimental Control in the Clinical, Affective and Social Neurosciences. *Frontiers in Human Neuroscience*, *9*. https://doi.org/10.3389/fnhum.2015.00660

Peirce, J. W. (2007). PsychoPy—Psychophysics software in Python. *J Neurosci Methods*, *162*, 8–13.

Peirce, J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics*, *2*. https://www.frontiersin.org/articles/10.3389/neuro.11.010.2008

Peirce, J. W., Gray, J. R., Simpson, S., MacAskill, M., Höchenberger, R., Sogo, H., Kastman, E., & Lindeløv, J. K. (2019). PsychoPy2: Experiments in behavior made easy. *Behavior Research Methods*, *51*(1), 195–203. https://doi.org/10.3758/s13428-018-01193-y

Pelli, D. G., & Tillman, K. A. (2008). The uncrowded window of object recognition. *Nat Neurosci*, *11*, 1129–1135. https://doi.org/10.1038/nn.2187

Qian, Y. Y., & Teather, R. J. (2017). The eyes don’t have it: An empirical comparison of head-based and eye-based selection in virtual reality. *Proceedings of the 5th Symposium on Spatial User Interaction*, 91–98. https://doi.org/10.1145/3131277.3132182

Rizzo, A. “Skip,” Goodwin, G. J., De Vito, A. N., & Bell, J. D. (2021). Recent advances in virtual reality and psychology: Introduction to the special issue. *Translational Issues in Psychological Science, 7*(3), 213–217. https://doi.org/10.1037/tps0000316

Scarfe, P., & Glennerster, A. (2015). Using high-fidelity virtual reality to study perception in freely moving observers. *Journal of Vision*, *15*(9), 3. https://doi.org/10.1167/15.9.3

Schloss, K. B., Schoenlein, M. A., Tredinnick, R., Smith, S., Miller, N., Racey, C., Castro, C., & Rokers, B. (2021). The UW Virtual Brain Project: An immersive approach to teaching functional neuroanatomy. *Translational Issues in Psychological Science, 7*(3), 297–314. https://doi.org/10.1037/tps0000281

Schreiber, K. M., & Schor, C. M. (2007). A virtual ophthalmotrope illustrating oculomotor coordinate systems and retinal projection geometry. *J Vis*, *7*, 4 1-14.

Soans, R. S., Renken, R. J., John, J., Bhongade, A., Raj, D., Saxena, R., Tandon, R., Gandhi, T. K., & Cornelissen, F. W. (2021). Patients Prefer a Virtual Reality Approach Over a Similarly Performing Screen-Based Approach for Continuous Oculomotor-Based Screening of Glaucomatous and Neuro-Ophthalmological Visual Field Defects. *Frontiers in Neuroscience*, *15*, 745355. https://doi.org/10.3389/fnins.2021.745355

Tachibana, R., & Matsumiya, K. (2021). Accuracy and precision of visual and auditory stimulus presentation in virtual reality in Python 2 and 3 environments for human behavior research. *Behavior Research Methods*. https://doi.org/10.3758/s13428-021-01663-w

UNESCO Recommendation on Open Science—UNESCO Digital Library. (n.d.). Retrieved February 17, 2023, from https://unesdoc.unesco.org/ark:/48223/pf0000379949.locale=en

Van-Roy, P., & Haridi, S. (2004). *Concepts, techniques, and models of computer programming*. MIT Press.

Wiesing, M., Fink, G. R., & Weidner, R. (2020). Accuracy and precision of stimulus timing and reaction times with Unreal Engine and SteamVR. *PLOS ONE*, *15*(4), e0231152. https://doi.org/10.1371/journal.pone.0231152

Wiesing, M., Kartashova, T., & Zimmermann, E. (2021). Adaptation of pointing and visual localization in depth around the natural grasping distance. *Journal of Neurophysiology, 125*(6), 2206–2218. https://doi.org/10.1152/jn.00012.2021

Wilson, C. J., & Soranzo, A. (2015). The Use of Virtual Reality in Psychology: A Case Study in Visual Perception. *Computational and Mathematical Methods in Medicine*, *2015*, 1–7. https://doi.org/10.1155/2015/151702

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., & Wilson, P. (2014). Best Practices for Scientific Computing. *PLOS Biology*, *12*(1), e1001745. https://doi.org/10.1371/journal.pbio.1001745

Yu, D., Liang, H.-N., Lu, F., Nanjappan, V., Papangelis, K., & Wang, W. (2018). Target Selection in Head-Mounted Display Virtual Reality Environments. *JOURNAL OF UNIVERSAL COMPUTER SCIENCE*, *24*(9), 1217–1243.