**Detailed protocol implementation in immersive Virtual Reality**

1. **Equipment and software**
   1. **iVR Equipment**

**iVR system:** The iVR system used was the HTC VIVE technology which comprises VR headset, cameras called base stations and a set of controllers. VR headset display offers a 360° field of vision (120° horizontal, in a stable position), with a refreshing framerate at 120Hz, which translates into a smooth display of movement. Pixel resolution goes up to 2448x2448 pixels per eye-screen. HTC VIVE headset and controllers are compatible with SteamVR® 1.0 and 2.0 base stations which detect the headset position (user’s position) and configurate the field of movement available. Cameras are placed in all corners of the room to define the movement space range: 10m x10m range of movement with four base stations and 5mx5m range with two stations (see details on product: <https://www.vive.com/fr/product/vive-pro2-full-kit/specs/>).

**Computer settings:** Robust computer requirements are needed for correct iVR functioning.The processor used was an Intel® Core™ i7-7700K CPU at 4.20GHz with the NVIDIA GeForce GTX 1060 6GB graphic card under Windows® 10 exploiting system.

* 1. **VR software**

The environment context was created in VR. Two software were used to model and build the environment: Blender2.92 ® and Unity 2018.2.16f1 ®.

**Blender ®** is a free and open software offering a wide range of tools for 3D creations in particular the following used to create the elements of our environment: modeling, rigging, rendering, compositing and texture painting.

**Unity ®** is a free and open game engine platform developed by Unity Technologies, [www.unity3D.com](http://www.unity3D.com). This software is easy to use for beginners and provides a full-range of tutorials to learn proper usage. It is mainly used for 2D and 3D video-game development. For our protocol, the 2018 Unity version number 2.16 was used to construct the virtual environment with the modeled elements in Blender as well as program movement, effects and over-all time line of the experiment. Coding is done in C# on the Visual Studio platform which is directly plugged in Unity.

**Visual Studio** is a developing platform software which comprises a set of tools that uses different coding languages such as Visual Basic, Visual C++ and Visual C#. Every coding language share the same *integrated development environment* (IDE) which is a set of tools and database that enables to enhance coding productivity for software-creating programmers by automating events and simplifying processes.

Within the framework of Unity, we use C# coding and appeal to various database such as: *System.Collections*, *System.collections.Generic* and *UnityEngine.*

For our virtual environment, we mainly scripted *classes of objects*. In Visual Studio an object is a combination of code and data that are handled as a single item. Every object is defined by a class. A class describes variables, properties, procedures and events of an object. Objects are instances of a class. Once a class is defined, as many objects as needed can arise from this class.

**HTC Vive plugin Unity ® :** To implement Immersive VR in Unity, the need to plugin Virtual glasses into the Unity project is essential to project the game screen as a 3-dimensional scene with a 360° overview. Unity asset store provides the VIVE Input Utility (VIU), which is a toolkit that enables developing immersive VR practices specifically with the HTC VIVE/VIVE Pro glasses. For more information on the package content check the [asset store website](https://assetstore.unity.com/packages/tools/integration/vive-input-utility-64219#description).

1. **Virtual environment**

The virtual environment is a three-roomed apartment composed of a kitchen, a living-room, a bathroom all separated by a central round-shaped hallway (figure 1).

Une image contenant capture d’écran, conception, exposition, intérieur

Description générée automatiquement

**Figure 1: Virtual environment** modeled in Blender**®** and Unity **®**, comprised of a central hallway, a kitchen, a living-room and a bathroom.

* 1. **Apartment conception**

To build the main shape of the apartment, simple 3D polygons were glued together in Blender**®**: a hollow cylinder long enough to create a rounded wall and two flattened cylinders for the ceiling and ground of the hallway; six flattened cubes to create the walls, ceiling and ground floor of the three different rooms. Once every polygon is placed correctly in Blender so to create the core frame of the apartment, the latter is uploaded in Unity**®** as one single object: from Blender the file is exported and named as a ‘. fbx’ file named ‘Core shape’ that is saved directly in the Unity Project’s **Assets folder** (figure 2C). The file is then introduced in the main **Scene** of Unity (figure 2B) (space in which environments are set up) attached to an empty *GameObject* named ‘APPARTEMENT’ in the **Hierarchy** section (figure 2A).

*GameObjects* in Unity are fundamental items that embody scenery, props and characters. When created, these items are empty as they merely serve as vessels for components which instrument their functionality. Here, by attaching the apartment polygon to a *GameObject* we can manipulate the latter as an object. The concept of *GameObject* allows to define a hierarchy between components in Unity (figure 2A). For example, if we define the core of the apartment as a higher order vessel, then every sub-item attached to the latter follow the same guidelines attributes to the apartment shape. Every *GameObject* and its sub-items can be modified via the **Inspector** section (figure 2D), in which transformations (position, rotation and scale), rendering, scripts, texture and physics attributes are defined.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

**Figure 2: Working tool windows in Unity. A.** Hierarchy section, this section defines *GameObjects* and sub-items hierarchy of the environment placed in the main workspace: Main Scene **B.** Main Scene: this section enables to construct the scenery of the environment by introducing and positioning the different items selected from the Assets folder **C.** Asset’s folder: this folder contains every file needed for the environment (blender imported files, textures, materials, scripts, plug-ins, packages) which are modified in the Inspector section **D.** Inspector section: this section enables to read and/or transform, define and edit every item imported into the asset’s folder.

Each room except for the hallway is composed of a numerous amount of furniture allowing to recognize its function (table 1). The furniture is created in Blender**®** and imported into Unity**®** by following the same procedure described here above.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**Table 1: List of furniture** modeled in Blender**®** for each room: living-room, bathroom and kitchen. Each furniture item is imported in Unity**®** as a *GameObject.*

* 1. **Object fabrication**

A total of 153 objects were modeled in Blender**®** (51 objects per-room). Although prefabricated objects can be found directly in the asset store of Unity, the latter does not offer this amount of variety in the repertoire of room specific objects, nor are all items cost-free. The goal here was to not only create our own personalized objects but also optimize their realism and identification. The list of objects is given in table 3

Une image contenant texte, capture d’écran, menu, Police

Description générée automatiquement

**Table 3: List of 153 objects modeled in Blender ®** 51 objects per room (living-room, kitchen and bathroom)

Objects were selected following the Bank of Standardized Stimuli (BOSS, Brodeur et al. 2014). The BOSS normative databank classifies visual stimuli of objects according to their percentile of *no-recognition,* called *Don’t Know Object* percentile *(DKNO%)* (number of no recognition/number of input) and percentile of *no-naming,* called *Don’t know Name* percentile *(DKN%)* (number of no name given/number of input).

We based our selection primarily on DKO% as recognition of objects was our priority. Objects for which DKO% was less than 20 percent were chosen for our environment out of which 68.6% percent have a DKO% equal to 0 i.e. systematically identified objects (82% of kitchen-objects, 75% of living-room objects, 49% of bathroom-objects) (table 3). Overall, mean DKO% was inferior to 3% for each room. However, out of the 51 objects created per room, some did not appear in the BOSS normative databank: 3 of the kitchen objects, 7 of the living-room objects and 12 of the bathroom objects. To verify correct identification of these objects, the latter were tested during pilot phase of the study (see “iVR testing” below). Details of Objects BOSS norms are given in supplementary data.

Une image contenant texte, nombre, Police, capture d’écran

Description générée automatiquement

**Table 3: Mean BOSS norms** for selected objects for each room of the apartment (kitchen, living-room, bathroom). Don’t Know Objects percentages (DKO%) are of 1.8%, 1.6% and 3% for the kitchen, living room and bathroom respectively. DKN%: Don’t know Namepercentile.

1. **Experimental flow implementation**
   1. **Avatar movement**

An avatar was created to set a reference for the subject’s camera view. This avatar was elaborated via the software Makehuman ® – a plugin for Blender ®, which main use is to model Avatars with the following creation pipeline: modelling (age, sex, race, body proportions), topology, materials, animation and rendering. The folder is then exported as a “. mhx” file to Blender which offers further modelling tools. In blender, the avatar’s position, clothes, bone placement, texture and animation can be modified.

In the said blender file, a long-blue lounge chair was modeled to match as identically as possible the real one where users lay on during immersion (figure 3). The avatar is modeled as seating on the chair by bending its bones’ appointment. The idea is to recreate entirely the position of the user so to better immersion in the environment.

Once the avatar and its chair were finalized, the file was exported and imported into Unity as a sub-item of the main Camera in Unity (first person focal view). The avatar/chair montage was placed in the center of the hallway as an initial neutral position.

Une image contenant capture d’écran, bleu, chaise, conception

Description générée automatiquement

**Figure 3: Real vs Virtual positioning on** lower-left user’s position on the blue chair (upper-left) vs. on lower-right, virtual avatar’s position on the modeled blue chair (upper right) in the apartment central hallway.

In Unity, scripts can be created and coded on Visual Studio, which are then affected to specific *GameObjects* of the scene to enable basic movements, tasks and overall experimental flow implementation. The following set of scripts was written to define the movement of the avatar in each room of the apartment:

* **‘MovementManager’ class:** This script defines the movement class type. It describes the characteristics of each basic shift such as moving forwards, moving backwards, turning right, turning left and stop moving/rotating. Note that every class is composed of a *start* instruction (condition for which the class is activated) and an *update* instruction (update called per frame, with a resolution of 30 fps) (see ‘Movement Manager’ script in repository).
* **‘Cameraa’ script:** This class serves to link the functions created in ‘MovementManager’ to a GameObject in unity. In our case, we named it ‘Cameraa’ as it will be set on the Camera GameObject. To do so, we initiate the function with the instruction to call the component of the MovementManager Class. In the Update section, we defined precisely the different existing shift voids that activate an Object: *MoveForward, MoveBackward, StopMove, TurnLeft, TurnRight, StopRotation.* (see ‘Cameraa’ script in repository)
* **‘InputManager’ script:** This class describes the trajectory of the avatar in the environment. As the avatar/chair montage is a sub-item of the main camera, this script is affected to the latter and not the avatar itself. *Note that this script does not necessarily use every function defined in the two previous classes* (see ‘Input Manager script in repository).

if (timeleft < 48.5)

{

if (timeleft > 47.5)

{

this.transform.localRotation = Quaternion.AngleAxis(-580, Vector3.up);

}

else cube.StopRotation();

}

if (timeleft < 47.5)

{

if (timeleft > 41.5)

{

cube.MoveForward();

}

else cube.StopMove();

}

if (timeleft < 37.5)

{

if (timeleft > 36.5)

{

this.transform.localRotation = Quaternion.AngleAxis(-670, Vector3.up);

}

else cube.StopRotation();

}

if (timeleft <36.5)

{

if (timeleft > 31.5)

{

cube.MoveForward();

}

else cube.StopMove();

}

if (timeleft <27.5)

{

if (timeleft > 26.5)

{

transform.localRotation = Quaternion.AngleAxis(-760, Vector3.up);

}

else cube.StopRotation();

}

if (timeleft < 26.5)

{

if (timeleft > 22.5)

{

cube.MoveForward();

}

else cube.StopMove();

}

if (timeleft < 21.5)

{

if (timeleft > 20.5)

{

transform.position = Cube3.transform.position;

transform.localRotation = Quaternion.AngleAxis(-760, Vector3.up);

}

else cube.StopRotation();

}

Instruction of frame-per-frame update describe one circuit of the avatar in the apartment. Note that, in this example of circuit, nor the preprogrammed rotation functions (*TurnRight*, *TurnLeft*) neither the random rotation (*randturn*) were used for the purpose of the aimed protocol. Instead, were used precisely defined instant rotations, called *local rotation transformation,* in terms of both direction and amplitude, which the user sees as a jump from one point of view to another. The update function first sets the time countdown: t*imeleft -= Time.deltaTime* decreases the *timeleft* variable of one frame per past frame. The different phases of the circuit are then based on its value.

The avatar is first positioned in the hallway center facing the first door to be crossed. The starting time is *timeleft=225s* and stop time *timeleft=-2.5s*, meaning a duration of *227.5s (3.8min)* for one round around the entire apartment. The following steps describe one circuit in one room (figure 4):

1.The avatar starts moving forward towards the door for 3.5s and stops for 1.5s. In that stop time, the door opens (see door opening script described below).

2. The avatar moves forward for 5.0s to enter the first room and stops again for 4.0s.

3. The avatar then rotates instantly around the z-axis (called as Vector3.up) in the right direction at a 45-degree angle using the *local rotation transformation.*

4. The avatar moves forward for 3.0s and stops for 4.0s.

5. The avatar rotates again in the left direction at a 45-degree angle, moves forward for 6.0s and stops for 4.0s.

6.The avatar rotates instantly in the left direction at a 90-degree angle, moves for 5.0s and stops for 4.0s.

7. The avatar rotates instantly in the left direction at a 90-degree angle, moves for 5.0s and stops for 4.0s.

8. The avatar is instantly translated to a position in front of the door with a *position transformation* function and stops for 1.0s. In that stop time, the door opens (see door opening script described below).

9. The avatar Moves forward for 10.5s until reaching the center of the hallway again and stops for 10.0s.

10. The avatar rotates instantly so to face the next door.

That procedure is repeated 3 times so to travel once in each room of the apartment.

The entire circuit is a total of ten rounds around the apartment thus lasting 2275s (around 38 minutes) in total.

Une image contenant capture d’écran

Description générée automatiquement

**Figure 4: Programmed timeline of the avatar’s movement** around each room of the apartment (illustrated here in the living-room). Time count is based on number of frames (f) and types of movement include move forward, stop moving, instant rotation and instant translation.

* 1. **Environment movement**
* **‘DoorOpening’ class:** Doors were programmed to open and close according to the time lap of the avatar’s circuit.The script was based on the same time frame set up (*timeleft = 225.0f*) as the *InputManager* script (time frame of one round across the apartment). To open the doors, a simple rotation around the z axis was instructed in the correct time points (see ‘Door Opening #’ scripts in repository).
* **‘GlobalAppMovement’ class:** The apartment shape was conceived such as the user can not locate himself in the hallway. This enables to randomize the circuit in order to prevent early anticipation/expectation of each room. This way, the subject only realizes in which room the avatar will enter once the front door is opened. To do so, we programmed the rotation of every room around the hallway cylinder. The script *GlobalAppMovement* is based on a time countdown timeleft = 2275.0f, entire duration of the avatar’ circuit (ten rounds in each room in total) (see ‘Global App Movement’ script in repository).
  1. **Example of cognitive task development:**

An example of a cognitive task that can be used in this environment is a visual search task. Here we used simple geometric shapes that we placed on random objects. Seven different kinds of shapes were created (square, rectangle, triangle, star, hexagon, circle, diamond) all of color green. Subjects are instructed to carefully observe every object of each room so to find these shapes that can appear on any object. Each time the subject enters once again in a room, the task is reset.

To create this task, seven specifically shaped polygons were modelled in Blender® and introduced into Unity® as sub-items of seven randomly selected objects per-room. An added script to these objects enabled to make shapes appear/ disappear as desired according to the user’s circuit. To do so, the function ‘SetActive’ was used (function available in the database). This function is either set on *True* or *False* to activate or deactivate the appearance of a shape on the associated object. Here below is the script associated to the shapes of the kitchen objects called *SearchTaskKitchen* (see ‘Set Active’ script in repository).

Une image contenant capture d’écran, intérieur, conception

Description générée automatiquement

**Figure 5: Illustration of the Search task** in the kitchen (left) and the living-room (right). Green shapes include square, rectangle, triangle, star, hexagon, circle, diamond.

* 1. **Launching interface**

**Main menu:** A main menu interface was created composed of two buttons: *play* and *quit* buttons (see ‘Main Menu’ script in repository). The ‘MainMenu’ script uses Unity’s *SceneManagement* toolbox which enables to load and close scenes created in the Unity project folder. The play button loads the main apartment scene with the *PlayGame* function, and the quit button associated to a *QuitGame* function closes the application (see Application for running below).

Une image contenant texte, capture d’écran, Police, blanc

Description générée automatiquement

**Figure 6: Main Menu interface**

**Quit button:** A simple quit button was added in the apartment scene, enabling to stop the application while environment is running (see ‘Quit Button’ script in repository).

**Application for running:** Once the entire modeling project is complete in Unity, the software enables to generate an application for simple launching of the environment. To do so, application building settings allow selection of scenes to be built (MainMenu and Apartment scenes) and configurates the support system in which the application will run (target platform - Windows, architecture -x86\_64, compression method - default).