

ISS Erasmus Report

Student technical report – “Interactive Simulation Systems” course

Manuel Hernández Nájera-Alesón

January 29th, 2018

Supervisor: Siniša Popović

Abstract – Memory development under stress is a useful capacity for human beings. In this project It will be measured through a demanding task.

1 Introduction

Spatial memory has always been crucial for humankind when performing either 2 or 3-dimensional work. We depend heavily on our way of understanding and interacting with space and, of course, our way of remembering It's setting.

The aim of this project is to evaluate how different human beings can interact with 2D environments while under a situation of sudden stress caused by a fast-paced task accompanied by loud and screeky noises.

The individual will switch from a tranquil and low-demanding environment, in which they will be able to walk and see around the formerly mentioned task, which they will have to complete in a limited amount of time under extremely harsh circumstances.

This will allow us to measure the individual's capability to handle sudden and stressful situations (e.g. surgery, aviation) and their efficiency whilst performing it.

There are some examples of spatial working memory tests [1].

The project has been developed with Unity 2017.2.0f using scripting language C# and Visual Studio IDE. .NET Socket communication was used to transmit the data from unity to the server.

2 Unity approach

When thinking about the best approach to this project realization, It came out that an already-built game-development-engine would be the best option. After considering other alternatives, Unity was the most suitable option.

Unity is a multiplatform videogame engine, available for Microsoft Windows, OS X and Linux. It provides the developer with a compilation support with a wide range of platforms, a graphical engine, a physical engine and a complex object hierarchy.

Unity was the best approach to this problem because it consist's of a set of scenes that can be loaded one after another until the end of the game execution. This setting suited our need of having two diametrically oposed enviroments (one calmed, the other one stressing and demanding) by implementing each of them into a diferent scene.

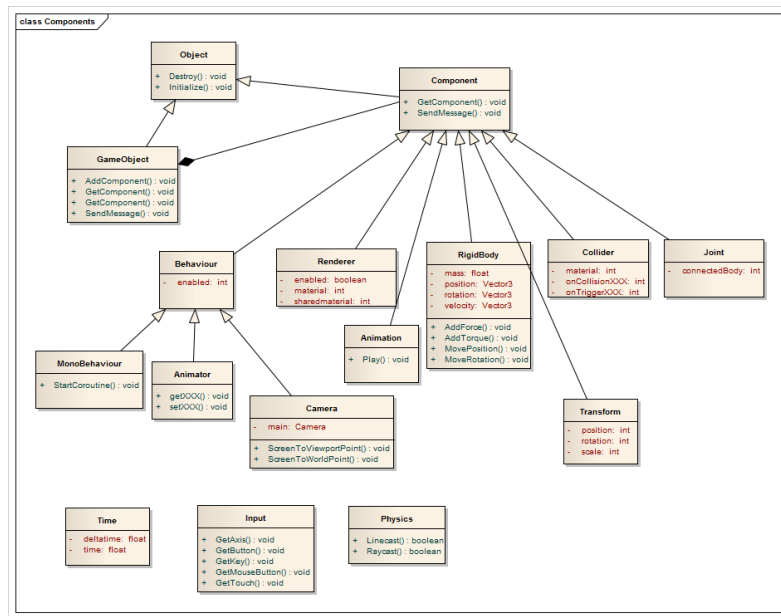


Figure 1. Basic Unity GameObject structure example. [1].

3 General Project Architecture

The general architecture of this project is resembled in figure 2. We have a Unity program running, which will take the parameters we will give it in a Setup file. With those parameters we will generate our program, with it's scenes and record the users input (clicks made). Then, with a TCP server created during the execution of the Unity Program, we will send the data to a TCP Client, that will output the data to the system console.

So, summarizing we have 2 types of communication involving the Unity program: TCP communication and Setup File communication.

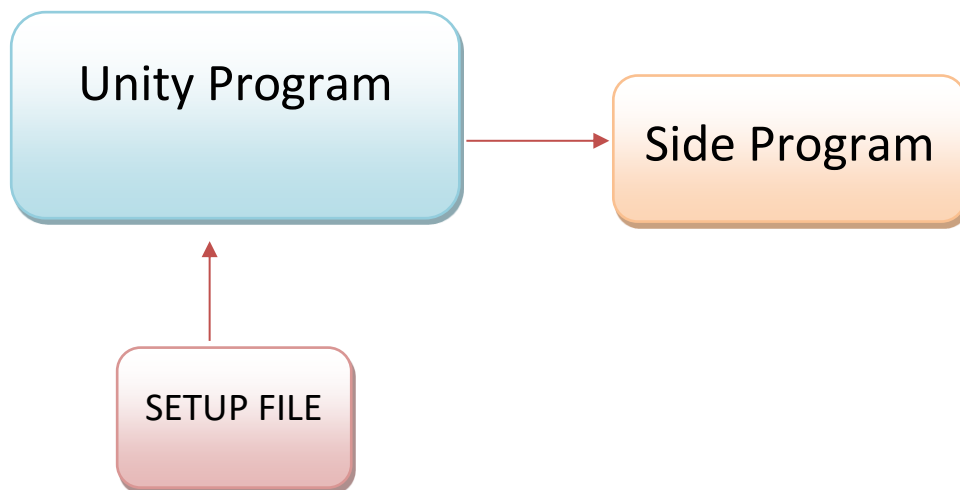


Figure 2. Basic Project Architecture

3.1 TCP Client and Server

When facing to the communication between the side program and the unity process, .NET socket communication was the option chosen.

TCP Server is run by a parallel thread of the Unity program, and has methods to be notified if one scene starts or ends or even send a string as a message. The Server has also a buffer (implemented as a double linked list), so

when the main unity program wants to send something to the client, it stores that message in the buffer. TCP Server will perform a buffer read from time to time, and send any information in the buffer.

TCP Client is a Side process that is executed upon the start of the Unity Program. The lifespan of this process is bound to the main program's lifespan, as it is closed when receiving an end message.

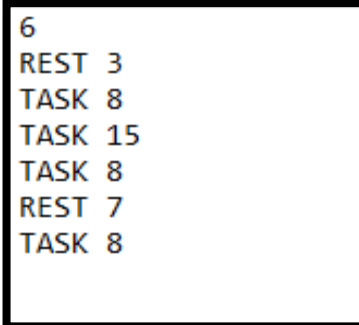
Communication between these two processes is made using socket communication through the local IP address of the machine, and is transmitted by bytes. When the TCP Server starts reading the buffer, it parses every byte of the string and sends it through the socket. When those bytes reach the client, a 2-phase reading takes place:

- Firstly, the 4 prior bytes of the string are read and stored. If the 4 of them are equal to **0xFF** that means we just received a 'Stop' signal, so the rest of the reading is aborted and the client proceeds to close.
- If former condition is not met, then we continue with the read and parse it to string. Proceeding then to write it in the console.

Communication is implemented in a way that, if another, more complex, side application is needed, the communication would not be a problem to reach.

3.2 File-Unity Communication

The communication between the task-programmer and the application is made via file reading. In this way we can configure how tasks and rests are scheduled.



```
6
REST 3
TASK 8
TASK 15
TASK 8
REST 7
TASK 8
```

Figure 3. Example of how the setup text file should look like.

The file is found in ".../Assets/Files/" and its name should be "SetupFile.txt". The organization of this file is as depicted in figure 3. First, an integer N which value is the number of lines after its line. Then we have the actual setup. N setup lines can be written. In setup lines first, it must be written either "TASK" or "REST" depending on the Unity scene we want to load. After that, a space and an integer must be written, representing the number of total seconds we want the scene to be active before changing to the next one.

4 Main Program Architecture

Unity main program is divided into scenes. This scene is act as separated workspaces and graphical environments. It must be considered that we are working with Object Oriented programming, written in C# Scripts. Typically, data from one scenes' objects is not shared unless we declare an object as non-destructible after the scene finishes. In this project that way of working had to be used and will be explained afterwards.

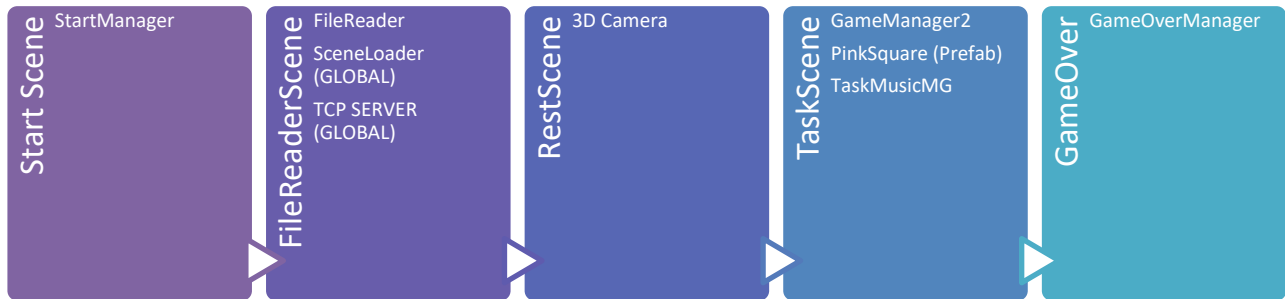


Figure 4. Object Organization in each scene of the project

This project consists of a set of 5 scenes:

4.1 Start Scene

Simple Scene in which we start with a black screen and white text in it telling the user to press spacebar to start the test. This Scene Jumps to “FileReaderScene”. This scene contains 1 script: StartManager, that implements the input reading and progressing to the next scene.

4.2 FileReaderScene

This is a transitory, yet important scene. During the execution of this scene, three scripts will be executed:

- **FileReader:** Attached to FileReader GameObject it reads the file mentioned in section 3.2. It loads a bi-dimensional array. First row’s values are either 0s or 1s, 0 for rest and 1 for task, the second row is filled with the time value of the *ith* row in the file. On finishing read, this array is passed to SceneLoader object.
- **SceneLoader:** Created at the start of the scene, it is not destroyed after the scene changes, it is only destroyed when the execution of the Unity program is finished. It stores the task/rest time array. This script is also responsible for scene switching according to the values stored. It performs this task by using the Unity method *Invoke()* which allows the user to make a delayed call to a function of their choice after a given time.
- **TCPServer:** The Object containing this script is also created at the beginning of the scene, and it is not destroyed after the scene concludes. After its creation, it sets up all the parameters for creating a server and then starts a parallel thread to run that server. (That way, Unity thread will not be stopped when the server is waiting for connection or to send information). The object is killed when execution ends and so is the thread, by a control variable stored in the main unity thread.

4.3 Relax Scene

In this scene the user will be able to walk and look around through a calmed environment consisting of a plane. W, A, S, D are the keys to move. If the user wanted to look around they should press the left button of the mouse then move the mouse to look around.

4.4 Task Scene

This scene consists of two parts. First, a set of some pink squares will appear for a time IT and the user will have to remember their positions. Then, those squares will disappear, and the user will have to click on the positions of those squares that they remember. Let it be noted that the program will only take into account the first N clicks, being N the number of squares in the scene.

To calculate the number of squares in the scene, a parametric equation is used.

$$T = N \times MCT + IT$$

Being T the duration of the scene (which is read from the setup file), MCT the Move and Click Time (time used to go from one square to another) and IT the Initial Time to look at the squares.

Therefore, the number of squares (N) is:

$$N = \frac{T}{MCT} - IT$$

MCT 's and IT 's values are predetermined and can be changed in the Unity Project.

Each time the user clicks two things can happen:

- They can click inside a pink rectangle, which will introduce a message in the server buffer with a "Clicked" and the coordinates of the correct click.
- They can click outside. In this case, the program will compute the shortest distance to a square and introduce a message in the server buffer containing "fail", where in the world the click was failed and the smallest heuristic distance to a square.

Squares are dispersed in the screen using a pseudo-random algorithm. The algorithm in charge of placing each square is of utmost importance, as in this project two conditions had to be reached for the square generation:

- First, no square could overlap other square. For this would could either confuse the user, leading them to think there is more than one shape or it could make the task easier for them, as clicking two times in the same spot would mean clicking correctly in a square two times without any visual memory effort.
- Second, no square should be critically close to other one, as it would, as well, make things easier for the user taking the test. Note that this concerns only squares being extremely close. That is, less than 10px of difference.

The difficult part of this task was to place every square checking if it was overlapping with some other square already in place.

Firstly, it was thought to approach this challenge with a 2Dimensional tree structure, having all the squares of the scene ordered by x-y position. This way, the cost of checking the overlap would be low. However, this solution could not be put into function due to the complexity of the implementation.

As a second approach, an exponential algorithm was thought of. In it, the program would, each time placing a square, check overlapping with any other square in the scene. This solution was discarded because of its cost.

The algorithm that was finally implemented does not need of boundaries checking. The basis of this algorithm is dividing the X axis of the screen into $3N$ intervals (being N the number of squares in the scene). Then, randomizing the X position of each square inside that interval and randomizing its Y position inside the screen.

With this implementation, no boundaries checking is needed, as there is no possibility that two squares overlap their X positions, due to the interval separation. But it provides some randomness, as the X of each square can vary inside its interval, and the Y randomizes along the screen.

During the scene, some sounds will be played by the TaskMusicManager object. 3 sounds are now implemented through the use of Unity's AudioSource and AudioClip:

- Static noise: This noise loops until the scene is over. It is the sound of the static noise of an old TV.
- Screams: This noise is played every 3 to 6 seconds (randomly). It is a high-pitched sound of a person screaming loudly and anxiously.
- Shots: This sound is played every 2 to 5 seconds (with another random seed). The sound consists of a gun shooting loudly and near.

4.5 GameOver Scene

In this scene one object exists, GameOverManager. It is responsible for the closing of the whole application.

When the game reaches gameOverScene, a message is sent to the TCP server to close itself and its thread and the application is closed, calling each objects *OnDestroy()* function.

5 Conclusion

This project, although simple, provides a simple way to measure fast reactions in any user. It provides raw data for other programs to refine. It is an extendable program, thanks to unity and its object architecture and it is mostly program, that is, the tester can decide which times to implement in the simulation.

6 References

- [1] Wang Xuanyi, Unity verview of basic concepts [online]
<http://www.programering.com/a/MDO4UDNwATY>.
- [2] Cambridge Cognition, Spatial Working Memory
<http://www.cambridgecognition.com/cantab/cognitive-tests/memory/spatial-working-memory-swm/>

7 Appendix

The project is only in its first state, an alpha. Most of the sounds are not implemented yet. Neither it is the main relax scene, which should be implemented with 3d models of trees, grass and so. However, it serves as a foundation and provides the basic interface and tools to fulfill its task.