Bachelor's thesis

Information and Communications Technology

2022

Tuan Nghia Tran

Eye-tracking data visualization to analyze player behavior

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | 57 pages

Tuan Nghia Tran

Eye-tracking data visualization to analyze player behavior

The thesis project is a part of a sea captain training project. The purpose of this project was to research and develop an immersive way to stimulate maritime operations. By using VR and eye-tracking technology, the stimulation collected player's behavior data. The collected data can be analysed so that the player’s performance can be assessed.

The objective of this was to develop a tool for analysts to analyze the collected data. The data collected every frame from the game-play is raw data in different forms. It includes 3D coordinates such as eyes and player position, Vector3 of the gaze vectors, floating values of pupil sizes, focus distance, etc. With massive raw data of different types analysts face the challenging task of analyzing and comparing how the player behavior changes over time. So the thesis aimed to research and produce a tool to visualize all the raw data into analyzable data for analysts. The analysis methods used for this tool are graph, heatmap, and scan-path. The graph method handles the change of data over time. The heatmap presents how the player spends their time on specific areas, and show the size of the event as color while the scan-path method shows how the gaze travels through different objects over time. The tool built in C# programming language and Unity game engine. The result was a tool that visualized the collected data by utilizing the above methods. The tool received positive feedback from Calin Calbureanu-Popescu for fulfill his requiment and valuable for his further research. Furthermore, the tool will be improved, optimized, and built-in into the main project.

Keywords:Game development, VR, Eyetracking, Data Visualization, Varjo headset, Marisot, Heatmap, Scanpath, Graphing, C#, Unity Engine

Contents

Figures

List of abbreviations (or) symbols

2D: Two-dimensional

3D: Three-dimensional

CSV: Comma-separated values

Jpg: Joint photographic group

Marisot: Maritime Immersive Safe Ocean Technologies

Png: Portable network graphic

VR: Virtual reality

1 Introduction

After decades of development, the gaming industry has become the largest among all the other entertainment industries. With an estimated 3.2 billion gamers worldwide and around $178.73 billion[1] revenue in 2021, the gaming industry surpassed other traditional entertainment industries like Broadcast TV, Radio & Music, Mobile phone, and Newspapers & Magazines[2]. That is remarkable and impressive for a forty-year-old industry to achieve this title. To continue with this massive growth over the years, the technology to go with the games also has to be rapidly developed. It changed speedily from arcade to console, PC, and mobile games. Recently, the technologies that have the most significant influence and potential for the gaming industry's future are Motion Capture and VR. Especially VR is opening a whole new chapter for game development. By using hand-tracking and eye-tracking, the experiences given to the player are immersive and as realistic as possible. Furthermore, VR technology is not just applied to game development but also to many other industries like healthcare, education, etc. Many large companies invest money into developing VR environments to support their product and future development. The MarISOT project is no exception. This thesis project is a part of the MarISOT project whose aim is to research and develop an immersive way to stimulate maritime operation[3]. By transforming maritime operation into a VR world, the training program for seafarers, sea captains, etc will be held handily in the form of application. The objective of this thesis is to develop a tool for the analyst to visualize eye-tracking data in analyzable forms and forward it to a solution for grading the training performance. With the overwhelming amount of raw data, analysts have a hard time figuring out what is going on with player behavior over time. With a tool that provides them with visualized data, analysts can access an overview of player behavior throughout the gameplay along with critical occasions that stood out in the performance. Even though other studies about the same topic exist, the applications only work in 2D and static environments. In a dynamic environment like VR, the same theory will not work accurately anymore. Furthermore, it is hoped that the work carried out in

this thesis will in some way or another contribute to the development of VR entertainment games as well as VR education games.

The visualization tool uses three methods: graph, heatmap, and scan path. Each method is responsible for a type of data or special requirement from the analyst. The amount of data collected from the Varjo headset and Unity editor combined are over twenty different types of data. With the graph method, raw floating data can be represented accurately over time. The result enables the analyst to examine the in-game events based on a different timestamp. The author used Unity Editor to draw the graph on a canvas in 3D space ingame. The second method used in this project is the heatmap. By definition, the heatmap is a technique that shows the size of an event as color[4]. In this case, it shows the intensity and attention of the eye's gaze on different game objects. Using this method provides analysts with an overview of how player spend their time in some crucial areas.

The last method used in this tool is Scanpath. In short, "scanpath" is a sequence of eye motion creating a path through specific points/objects when playing[5]. The purpose of this method is to check if the player's eye fixations are landed on an important object or in the correct order. It helps observers specifically check how eye fixations move through some object rather than showing the area as a heatmap does. Collecting data and scanpath from an experienced sea captain, it will show the correct or roughly accurate path as a sample for grading. In other words, a heatmap helps the analyst check if the player pays attention to the right area and a scanpath is used to investigate what the player is doing in that area.

This thesis is structured as follows: Chapter 1 introduces the thesis objectives. Chapters 2 and 3 provide the background of research studies and the MarIOST

application. Chapter 4 discusses data and data collection. Chapter 5 presents the research of navigation in a 3D environment. Chapter 6 describes methods developed in this tool. Chapter 7 presents the outcome result from the tool. Chapters 8 and 9 are Discussions and Conclusion.

2 Background of research studies

As mentioned in the introduction, VR games will be the future technology of the gaming industry plus all other entertainment industries. There are more and more VR games and applications developing every day. The VR gaming industry's global revenue in 2020 is 1.1 billion $ and will continue to grow over the years. It is forecast to reach 1.4 billion $ in 2021 and 1.6 billion $ in 2022(1). It is a promising market for the gaming industry but, it also brings challenges. The new technology also comes with less support and studies theory for further development. Most of the theories and studies applying to the gaming industry nowadays might not be able to use in a VR environment. Eyetracking studies are one of those. The previous studies about eye-tracking are focused on a static environment like a 2D screen and monitor. The differences in basic principles between static and dynamic environments are massive, which leads to differences in the end result. To fill those gaps that previous studies left behind, this study will focus on collecting and displaying data in a dynamic environment.

There are some examples of studies about visualization for eye-tracking like Visualization of Eye Tracking Data: A Taxonomy and Survey[6] and Visualization of Eye Tracking Data[7] that are worth to be mention. The scenario in this game is in a VR environment, and the player will be moving around are interacting with dynamic objects. Let us look at Figure 1 as an example.

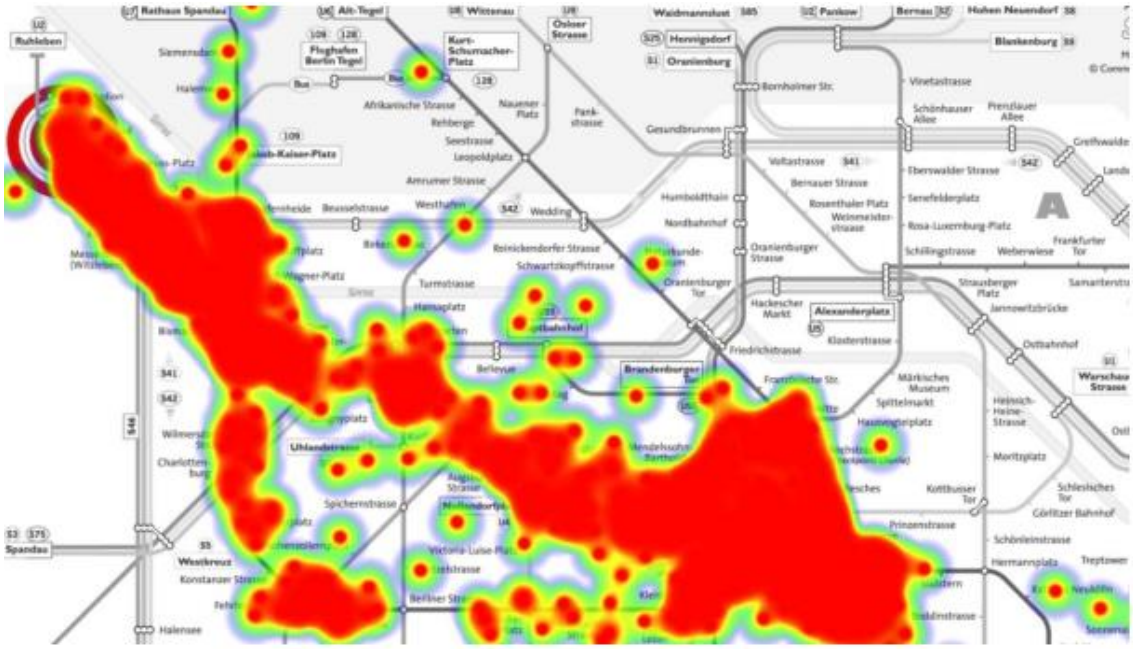


Figure 1. Example of 2D heatmap from other studies.

The heatmap was rendered on top of the screen capture. The colors speak for themselves. The warmer the color, the longer someone looks at it. The coordinate of fixation on the screen will be stored as the x and y coordinate of the screen. And, the color will be rendered based on that coordinate and displayed nicely on a 2D screen. Because of this, the application will be applied to a static environment like websites, Figures, videos, etc. However, let's put this into practice in a 3D environment and the difference is enormous. The fixations still land on the user screen and are collected into a database. Nevertheless, because it is a dynamic environment when the user changes their point of view by turning around, even though the fixation is still the same point on the screen, it is not the same object anymore. So the study was not valuable in this case and will display the result wrongly.

To be more specific, the previous study's outcome will be incorrect because of the lack of navigation inside the 3D environment. To fill the gap, the author created his navigation inside the VR environment based on data collection. It collects data on the player position as well as rotation related to the VR world. Because of player will be inside the VR environment and moving around locally

to it, the collected data will be continuous correctly regardless player's behavior. Because of that, navigation in a VR environment is crucial for this thesis and will be applied to all of the following visualization methods. By combining various data types collected from the Varjo headset, the author found a way to create his navigation in the VR world. Section Navigation goes on to the detail of the navigation research based on.

3 MarISOT application

MarISOT, short for Maritime Immersive Safe Ocean Technologies funded to develop an immersive maritime operations simulation. By transforming it into a VR world, the training program for seafarers, sea captains, etc will be held handily in the form of application. It leads to the fact that the cost and time for maritime operations are massive and not influential. Moreover, this solution will solve those problems by increasing performance while reducing the cost, also the effectiveness and safety of the training. The concept of this training program is to develop gameplay that which players will spawn in the command bridge and control the boat. The performance will be graded based on how safe the player drive and how well he follows the maritime instruction. To develop the scoring system, developers need to collect and analyze data from a veteran sea captain. After analyzing and researching what type of data will affect the overall performance like cognitive load and pupil sizes, the scoring system will depend on that and compare the difference with the player data to evaluate the final result. Furthermore, all the data will be studied by neural networks and based on that to develop an automatic grading system. This thesis provides a tool to visualize data for analysts to analyze player behavior. The data collected from players every frame is raw data with a massive amount. With this overwhelming amount of data, analysts have a hard time figuring out what is going on with player behavior over time. And, by visualizing each type of data, the analyst can observe how the data change over the gameplay time and spot some outstanding points.

The application is a VR application built in Unity Engine and uses the VR Varjo headset for the eye-tracking feature. Unity Engine was chosen because it is one of the most well-known game engines in the world and highly supports VR development. Other than that, the Unity game engine is primarily chosen for teaching and learning at Turku University of Applied Sciences as well as The Game Lab. Because of the following reason, Unity Engine is preferred for

project development. On the other hand, eye-tracking technology enables new forms of experience and interactions in a VR environment. This powerful technology provides complete insights for researching human behavior in the VR environment. And Varjo VR glasses is the high-tech VR glasses that support eye-tracking technology. With high resolution and a built-in library for Unity Engine development, the Varjo headset is a suitable choice for the study besides the project overall. Furthermore, the Varjo headset supports data collection in various types and is valuable for the research. Further discussion of data collection and its application of it will be in the next chapter.

4 Data

The Varjo headset is a high-tech VR glasses that have eye-tracking integrated. With the new technology that is built inside the headset, 37 different types of data have been collected by Varjo Base and extracted into CSV files. The next figure show extracted data from the Varjo headset in Excel form (Figure 2).

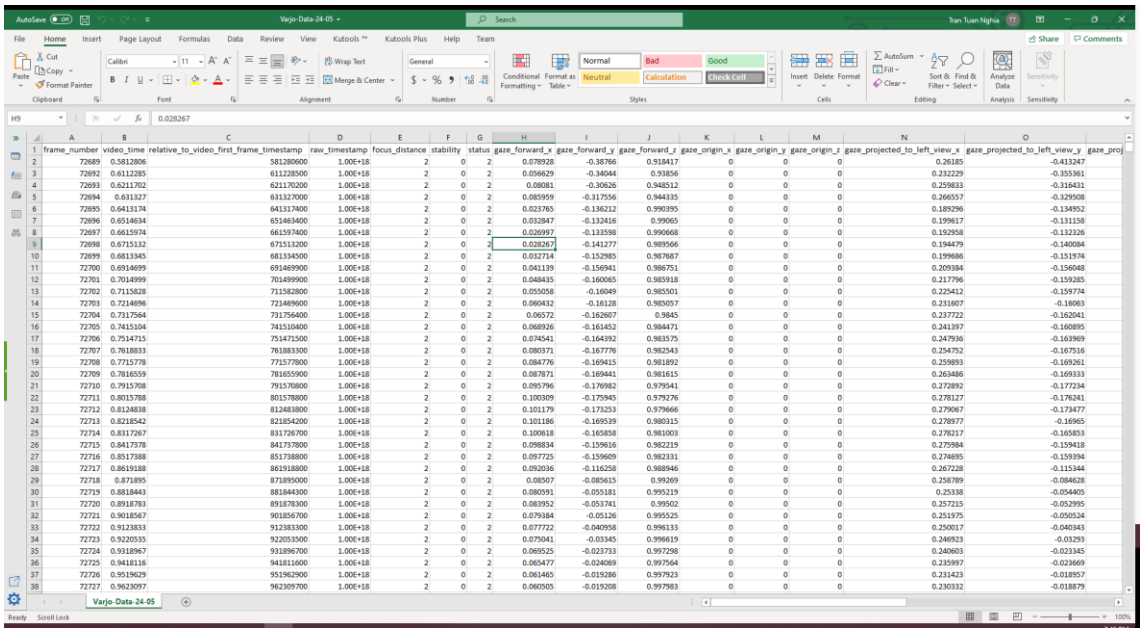
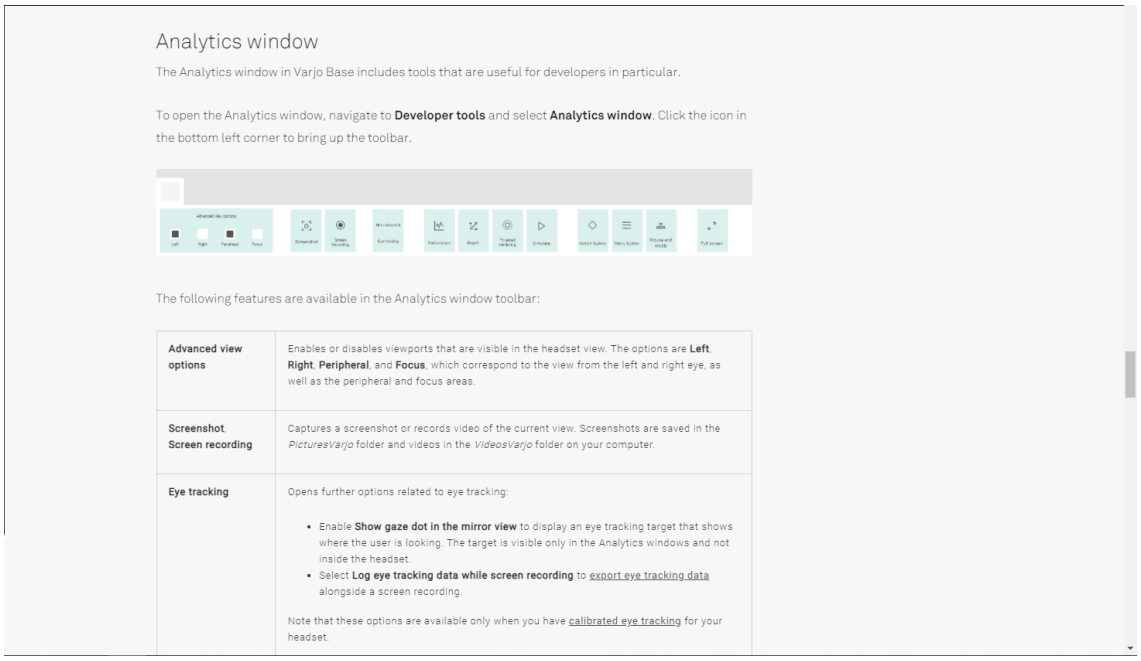


Figure 2. Varjo Base data.

To highlight, it included data about frames, video time, focus distance, stability, status, gaze vector, gaze origin, pupil sizes, etc. All of the data is in floating-point, and some of it needs to be combined to represent something else like 3D coordinate points and vectors. To visualize all of these raw data with such a massive amount, it needs different methods for a different use for each data. For floating data, the author combined it with timestamp data to represent the changing of value over time. Moreover, for this type of data, the graph method will be used to describe it. For 3D coordinate data like gaze position, gaze vectors, etc it will be used for navigation tools that help other methods like heatmap and scan-path. Other features from Varjo Base are also being used for

analytic progress. One of the standout features from it is Analytics Window (Figure 3) in Developer Tools.



Analytics window

The Analytics window in Varjo Base includes tools that are useful for developers in particular.

To open the Analytics window, navigate to **Developer tools** and select **Analytics window**. Click the icon in the bottom left corner to bring up the toolbar.

The following features are available in the Analytics window toolbar:

Advanced view options	Enables or disables viewports that are visible in the headset view. The options are **Left**, **Right**, **Peripheral**, and **Focus**, which correspond to the view from the left and right eye, as well as the peripheral and focus areas.
Screenshot, **Screen recording**	Captures a screenshot or records video of the current view. Screenshots are saved in the *PicturesVarjo* folder and videos in the *VideosVarjo* folder on your computer.
Eye tracking	Opens further options related to eye tracking: • Enable **Show gaze dot in the mirror view** to display an eye tracking target that shows where the user is looking. The target is visible only in the Analytics windows and not inside the headset. • Select **Log eye tracking data while screen recording** to export eye tracking data alongside a screen recording. Note that these options are available only when you have calibrated eye tracking for your headset.

Figure 3. Varjo Base analytics window.

These tools are built-in inside Varjo Base, it helps and saves plenty of time for developers to observe what is going on inside the headset. Advanced view options give the developer different points of view from the left and right eye, as well as peripheral and focus areas. Besides that, the combination of eye-tracking features and screen recording is valuable for analysts. By enabling the gaze dot screen recording, the analyst can have a better view of what is going on inside the headset and what the players are paying attention to and it can also be replayed multiple times for the purpose of analysis. Furthermore, the log eye-tracking data options are also beneficial for analysis purposes since they export eye-tracking data based on video timestamps. There are a couple more excellent features like performance, depth, etc, but they will not be used and covered in this thesis.

Besides extracting data straight from the headset by Varjo Base, there is also a way to extract the data from the game by using the VarjoPlugin library. The data collected from Unity has some slightly different. Other than frames, eyes gaze vectors, eyes status, focus distance, stability, and pupil size remains the same, Unity can collect some other data come from the game world. The data collected from Unity includes head position, rotation in world space, combined gaze forward also other custom data like game objects that player looks at. More information can be provided on player behavior in gameplay by customizing and extracting data from Unity. By using the VarjoPlugin library, the author can access gaze data and extract it into a log file (Figure 4).

```
void LogGazeData(VarjoPlugin.GazeData data)
{
    // Get HMD position and rotation
    hmdPosition = VarjoManager.Instance.HeadTransform.position;
    hmdRotation = VarjoManager.Instance.HeadTransform.rotation.eulerAngles;

    string[] logData = new string[19];

    // Gaze data frame number
    logData[0] = data.frameNumber.ToString();

    // Gaze data capture time (nanoseconds)
    logData[1] = data.captureTime.ToString();

    // Log time (seconds)
    logData[2] = ((DateTime.Now.Ticks - now.Ticks) / (TimeSpan.TicksPerSecond)).ToString();

    // HMD
    logData[3] = hmdPosition.ToString("F3");
    logData[4] = hmdRotation.ToString("F3");

    // Combined gaze
    bool invalid = data.status == VarjoPlugin.GazeStatus.INVALID;
    logData[5] = invalid ? InvalidString : ValidString;
    logData[6] = invalid ? "" : Double3ToString(data.gaze.forward);
    logData[7] = invalid ? "" : Double3ToString(data.gaze.position);

    // Left eye
    bool leftInvalid = data.leftStatus == VarjoPlugin.GazeEyeStatus.EYE_INVALID;
    logData[8] = leftInvalid ? InvalidString : ValidString;
    logData[9] = leftInvalid ? "" : Double3ToString(data.left.forward);
    logData[10] = leftInvalid ? "" : Double3ToString(data.left.position);
    logData[11] = leftInvalid ? "" : data.leftPupilSize.ToString();

    // Right eye
    bool rightInvalid = data.rightStatus == VarjoPlugin.GazeEyeStatus.EYE_INVALID;
    logData[12] = rightInvalid ? InvalidString : ValidString;
    logData[13] = rightInvalid ? "" : Double3ToString(data.right.forward);
    logData[14] = rightInvalid ? "" : Double3ToString(data.right.position);
    logData[15] = rightInvalid ? "" : data.rightPupilSize.ToString();

    // Focus
    logData[16] = invalid ? "" : data.focusDistance.ToString();
    logData[17] = invalid ? "" : data.focusStability.ToString();

    // GameObject
    logData[18] = invalid ? "" : logHelper.CurrentGazeTarget;

    Log(logData);
}
```

Figure 4. LogGazeData function.

Because both of the collected data from the Varjo headset and Unity Engine are useful for the analyst in their way, combining them would create a more significant outcome. The combined data set can provide sufficient information for the analyst to do further research, including videos of gameplay and raw

data, and in-game data related to it. The final result is the joining between two sets of data based on the frame captured.

The most ideal and convenient for the analyst is to build a software like iMotions(Figure 5) where the data will be embedded in the videos and shown to the viewer based on the timestamp.

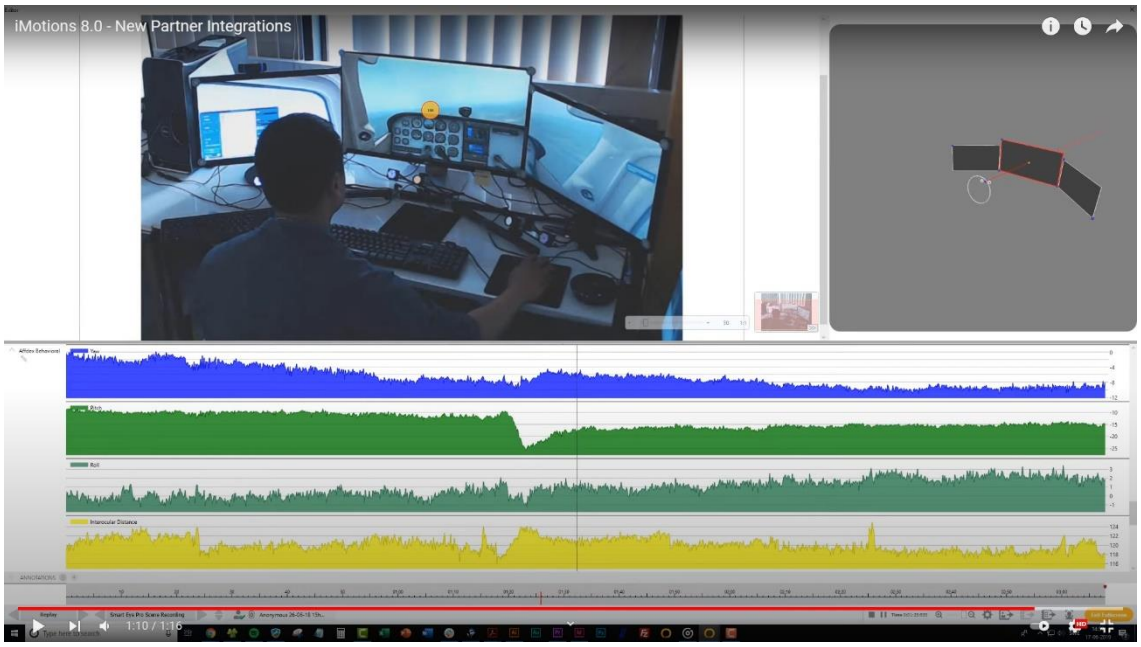


Figure 5. iMotions feature.

5 Navigation

Navigation is the most important determinant when correctly rendering a heatmap or scanpath or any kind of visualization method for a dynamic environment like VR. Because the point of view is changing all the time in the VR environment, the navigation can not be based on the screen coordinate. In this thesis, the author uses the correlation between player position and the environment around to navigate. The player spawns inside the virtual world, and the world is static. Even though the player moves around the environment nonstop but in the end, the player still stays inside it, and the environment remains the same. So instead of using screen position, which is changing dynamically, the environment position related to the player position is much more reliable. By casting a ray from the player position with the eyes gaze forward vector, it can detect what object the player is looking at in run-time. The source code in Figure 6 is what the author uses to navigate contact points with the environment by using the VarjoPlugin library[8].

```
void DetectEnvironmentWithVarjoGaze()
{
    if (!detecting)
        return;

    gazeData = VarjoPlugin.GetGaze();

    if(gazeData.status != VarjoPlugin.GazeStatus.INVALID)
    {
        gazeForward = new Vector3((float)gazeData.gaze.forward[0], (float)gazeData.gaze.forward[1], (float)gazeData.gaze.forward[2]);
        gazePosition = new Vector3((float)gazeData.gaze.position[0], (float)gazeData.gaze.position[1], (float)gazeData.gaze.position[2]);

        transform.position = VarjoManager.Instance.HeadTransform.position;
        transform.rotation = VarjoManager.Instance.HeadTransform.rotation;

        // Transform gaze direction and origin from HMD space to world space
        gazeRayDirection = transform.TransformVector(gazeForward);
        gazeRayOrigin = transform.TransformPoint(gazePosition);

        if (Physics.SphereCast(gazeRayOrigin, rayRadius, gazeRayDirection, out gazeRayHit, maxDistance, environmentLayer.value, collideToTriggers))
        {
            hitPoints.Add(commandBridge.InverseTransformPoint(gazeRayHit.point));
        }
    }
}
```

Figure 6. DetectEnvironmentWithVarjoGaze function.

Here is where the data that was collected before in use. By using GetGaze(), all information on gaze data will be returned, like in Figure 4. However, in this function, only the gaze origin position vector and gaze forward vector are

needed. And, because the vector value return is in the local space that is related to the Varjo head transform, it needs to be transformed to world space. As a result, the origin and direction for the environment detection are ready. From the SphereCastOrigin, it cast a SphereCast to the environment with the SphereCastDirection[9]. If it hits any environment object with the given layer, it will return the information of that collide. The value for this SphereCast info can be customized from the Unity's inspector (Figure 7).

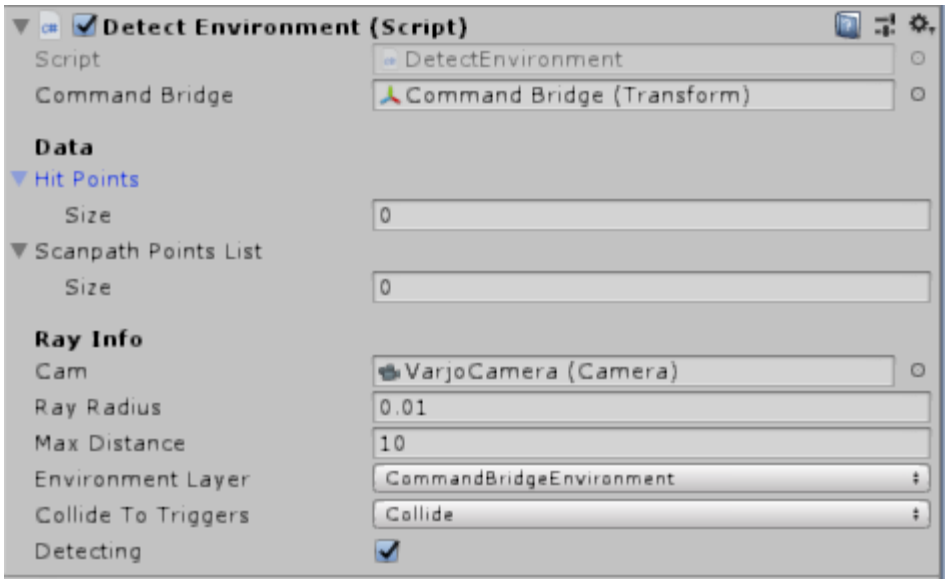


Figure 7. DetectEnvironment script variable from Unity Inspector.

After they collide, the detector identifies the environment object that the player looks at, the detector will collect the contact point and add it into an array to store. The contact point will be transformed to the command bridge local space. This is because the player moves locally inside the ship, but the ship itself will change its position in world space. If the player looks at any object on the ship like windows, control board, etc, from in local perspective, it is a static object, but it still changes its position over time in world space. So to keep the contact point precisely, it needs to be transformed to the ship's local space. Furthermore, all of these contact points are every position that the player's eyes gaze lands on and will be used later for rendering heatmap and scanpath.

6 Methods

The goal of this thesis is to build a tool that visualizes all the data collected into an analyzable form. As mentioned, this thesis used three methods: graph, heatmap, and scan-path to resolve those data.

5.1 Graph

This method is used to resolve the floating-point data and visualize it into value change over time. Furthermore, because the time used is frame time amongst all types of graphs, the line graph will be representing it precisely. The graph got two axes: the x-axis for time and the y-axis for data value. The name of the value will be on top, and the current object that the player is looking at is on the right of the graph (Figure 8).

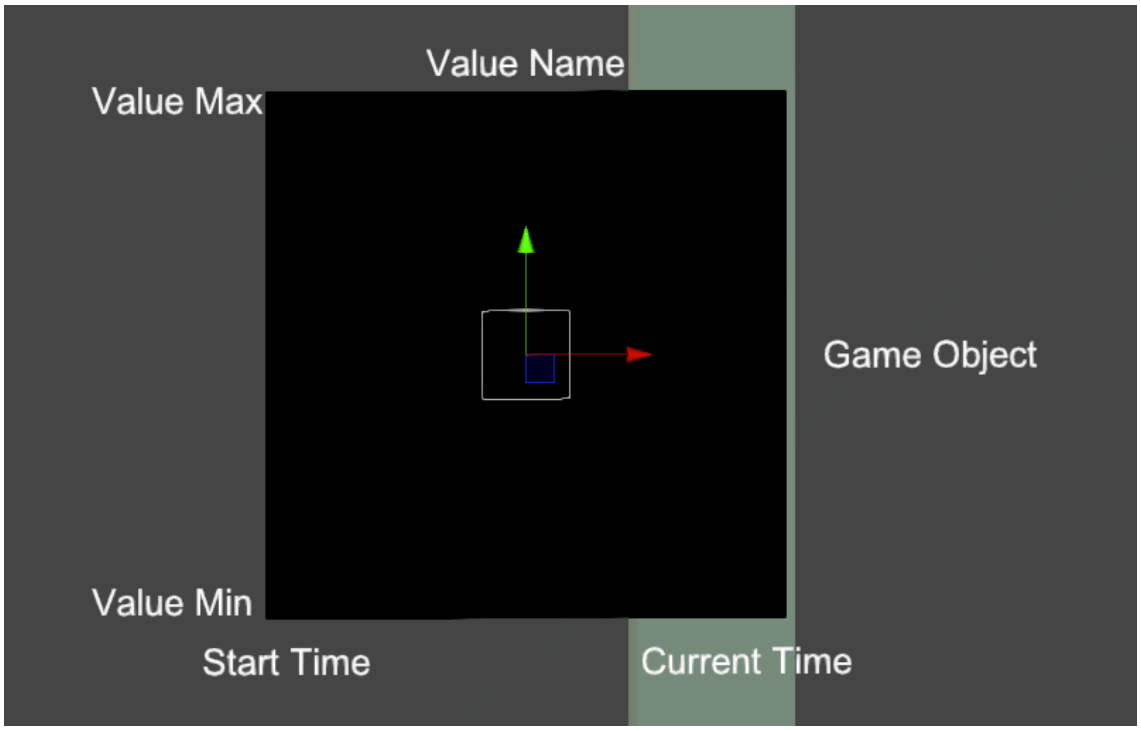


Figure 8. Graph layout.

When creating this graph in UnityEngine, first the creator added a black canvas and text reserved for the graph value. The value of those texts will be updated based on the change over time in the data. For the line, the creator used LineRenderer to render it on top of the black canvas in 3D space[10][11]. To contain all of the information needed for the graph, AnalyticGraph class was made. An AnalyticGraph object name valueGraph is created when the script starts, and the value for it will be initialized from the Unity's inspector(Figure 9).

```
[System.Serializable]
public class AnalyticGraph
{
    public Transform graphHolder;

    public Canvas graphCanvas;

    public Text valueName;
    public Text valueMaxText;
    public Text valueMinText;
    public Text timeStartText;
    public Text timeCurrentText;
    public Text gameObjectText;

    [HideInInspector] public LineRenderer line;
}
```

Figure 9. AnalyticGraph class.

GraphManager script was created to manage the graph (Figure 10).

```
void SpawnLine(AnalyticGraph graph)
{
    GameObject analyticsLineClone = GameObject.Instantiate(analyticLinePrefab, graph.graphHolder.position, graph.graphHolder.rotation, graph.graphHolder);
    graph.line = analyticsLineClone.GetComponent<LineRenderer>();
}

public void ChangeValueGraphName(string name)
{
    valueGraph.valueName.text = name;
}

public void ChangeGameObjectText(string name)
{
    valueGraph.gameObjectText.text = name;
}

public void DrawCurve(AnalyticPoint analyticPoint)
{
    if (enableAnalytics == false)
        return;
    if (valueGraph.line == null)
        SpawnLine(valueGraph);

    // Set linerenderer position point = analytic point
    valueGraph.line.positionCount = analyticPoint.analyticsValues.Count;

    for(int i = 0; i < analyticPoint.analyticsValues.Count; i++)
    {
        Vector3 newPosition = Vector3.zero;

        // Set value for each position point
        if ((analyticPoint.maxValue - analyticPoint.minValue) != 0)
            newPosition.y =(float)((analyticPoint.analyticsValues.Values.ElementAt(i) - analyticPoint.minValue) / (analyticPoint.maxValue - analyticPoint.minValue));

        if (analyticPoint.analyticsValues.Keys.ElementAt(analyticPoint.analyticsValues.Count-1) != 0)
            newPosition.x = (float)(analyticPoint.analyticsValues.Keys.ElementAt(i) / analyticPoint.analyticsValues.Keys.ElementAt(analyticPoint.analyticsValues.Count - 1));

        valueGraph.line.SetPosition(i, newPosition);
    }

    // Set range of value for UI element
    valueGraph.valueMaxText.text = analyticPoint.maxValue.ToString("F1");
    valueGraph.valueMinText.text = analyticPoint.minValue.ToString("F1");

    valueGraph.timeStartText.text = "0";
    valueGraph.timeCurrentText.text = analyticPoint.analyticsValues.Keys.ElementAt(analyticPoint.analyticsValues.Count - 1).ToString("F1");
}

public void Clear()
{
    if (valueGraph.line == null)
        return;
    valueGraph.line.positionCount = 0;
    valueGraph.gameObjectText.text = "";
    valueGraph.valueName.text = "";
}
```

Figure 10. GraphManager subfunction.

It had five functions to handle the graph change. ChangeValueGraphName() is used to change graph name into given string. ChangeGameObjectText() change the text into the name of the game object that the player is looking.

The Clear() function will refresh all the values of the graph. DrawCurve () will handle the rendering for the line graph. The function will take an AnalyticPoint object as a parameter (Figure 11).

```
[System.Serializable]
public class AnalyticPoint {

    public string name;
    public Dictionary<float, double> analyticsValues = new Dictionary<float, double>();
    public double maxValue =1;
    public double minValue =0;
    public AnalyticPoint(string n)
    {
        name = n;
    }
    public void AddAnalyticValue(float time, double value)
    {
        if (value > maxValue)
            maxValue = value;

        if (value < minValue)
            minValue = value;
        analyticsValues.Add(time, value);
    }
}
```

Figure 11. AnalyticPoint class.

For each of the data that needs to be present, an AnalyticPoint object will be initialized for it. The analyticValues will be stored as a dictionary with the type of <float, double> by pairing the data value and the time it has been captured.

Let's get back to the DrawCurve() function. After the AnalyticPoint object gets passed to the function, it gets all the information needed to draw the line. First, it spawns the LineRenderer object to the canvas position by SpawnLine() function. And then set the number of vertices as the number of AnalyticPoint value set. The next step is to run a loop through every pair value given from the AnalyticPoint object. Calculating the ratio between the value and the min, and max value will determine the position of the line vertex in the graph. When the loop is finished, all the vertex positions have been added, the line is complete as well as the label text will be updated. The result will look like Figure 12. The graph is used to display data with float values over time like pupil size, focus distance, stability, etc. To observe another set of data, press the right and left arrow buttons on the keyboard to go back and forth for it. The input handler and collecting data are written in GraphObject script (Figure 13).

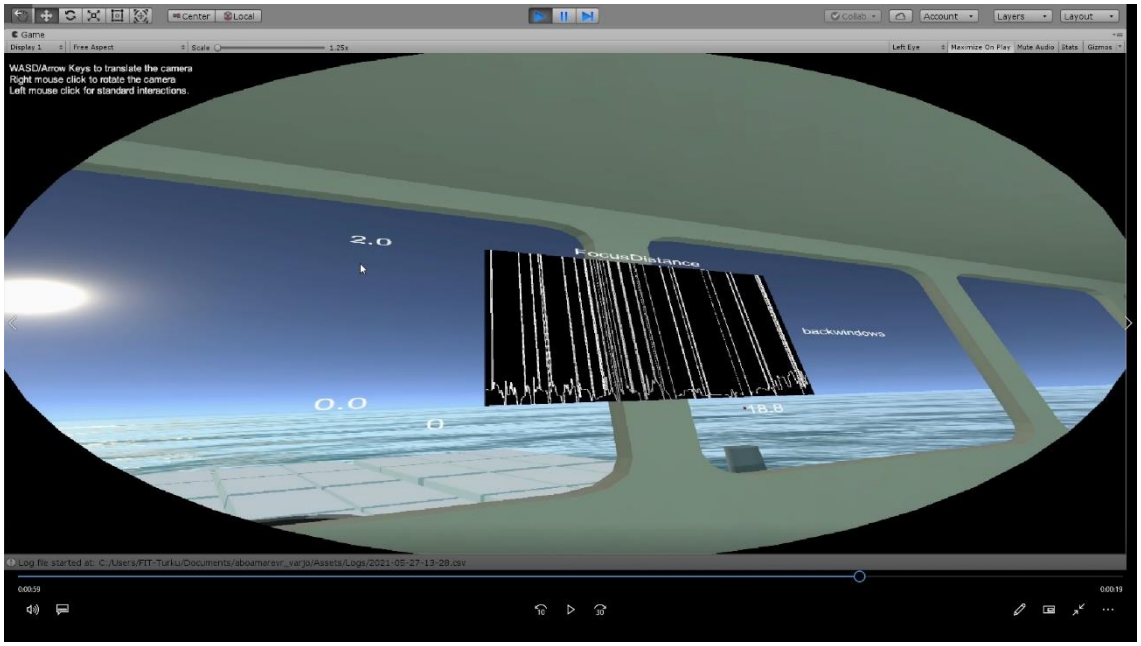


Figure 12. Graph in VR.

```
void HandleInput()
{
    if (Input.GetKeyDown(KeyCode.CapsLock))
    {
        StartCoroutine(CollectGazeData());
        StartCoroutine(CollectAnalytics());
    }
    if (Input.GetKeyDown(KeyCode.Escape))
        DisableAnalytics();
    if (Input.GetKeyDown(KeyCode.RightArrow))
        dataIndex++;
    if (Input.GetKeyDown(KeyCode.LeftArrow))
        dataIndex--;

}

IEnumerator CollectGazeData()
{
    double[] logData = new double[4];

    while (true)
    {
        // Left eye
        var data = VarjoPlugin.GetGaze();
        currentTime += Time.deltaTime;
        bool leftInvalid = data.leftStatus == VarjoPlugin.GazeEyeStatus.EYE_INVALID;
        logData[0] = leftInvalid ? 0 : data.leftPupilSize;

        // Right eye
        bool rightInvalid = data.rightStatus == VarjoPlugin.GazeEyeStatus.EYE_INVALID;
        logData[1] = rightInvalid ? 0 : data.rightPupilSize;

        bool invalid = data.status == VarjoPlugin.GazeStatus.INVALID;
        // Focus
        logData[2] = invalid ? 0 : data.focusDistance;
        logData[3] = invalid ? 0 : data.focusStability;

        int index = 0;
        foreach (AnalyticPoint a in analyticPoints)
        {
            a.AddAnalyticValue(currentTime, logData[index]);
            index++;
        }
        yield return null;
    }
}

IEnumerator CollectAnalytics()
{
    if (!enableAnalytics)
        yield break;

    while (true)
    {
        if (!enableAnalytics)
            yield break;

        GraphManager.instance.ChangeValueGraphName(analyticPoints[dataIndex].name);
        GraphManager.instance.DrawCurve(analyticPoints[dataIndex]);
        GraphManager.instance.ChangeGameObjectText(logHelper.CurrentGazeTarget);

        yield return new WaitForSeconds(analyticsTimeStep);
    }
}
```

Figure 13. GraphObject subfunction.

This script will collect data needed for the graph methods and call GraphManager to render the graph. Because the data has been collected in every frame so the amount of data is massive. So if the graph updates every frame, the script will be too heavy and affect the performance. Using the analyticsTimeStep as 0.01 second, the graph will be updated smoothly without affecting the performance. The tool was tested and well performed in the VR environment. The length of the test was about one to two minutes. However, because of the massive amount of data collected in every frame, the performance of the graph might go down over time. So the author recommends clearing and re-render the graph every one or two mins to guarantee a smooth performance. In the training program for the Marisot project, the gameplay might be much longer than this demo project.

5.2 Heatmap

The second method for this visualization tool is the heatmap. It will show how much attention players pay to an area through colors. The warmer the color is, the more time player spends on that area. This method can help the analyst in research which area is essential and how different it is from a regular person and an experienced one. For rendering, the heatmap will be rendered in a sphere with invisible material surrounding the player[12]. The color will be put on the material based on the player's gaze. Moreover, the final result will be extracted from a 360 image from the player's perspective. To archive this result, it needs to go through many small steps and take a closer look at it.

The HeatMapGenerate script will handle the rendering heatmap for this thesis. All the parameters with SerializeField will be initialized from Unity’s inspector (Figure 14,15).

```
public class HeatMapGenerate : MonoBehaviour {

    [Header("Visual parameter")]

    [Range(1, 10)]
    [SerializeField] private int radiusInfluence;

    [SerializeField] private Size TexResolution;

    [SerializeField] private GameObject sphere;

    [SerializeField] private Transform commandBridge;

    [SerializeField] private Gradient gradientFinal;

    [Header("RayInfo")]
    [SerializeField] Camera cam;
    [SerializeField] float rayRadius = 0.01f;
    [SerializeField] float maxDistance = 10f;
    [SerializeField] LayerMask heatmapLayer;
    [SerializeField] private QueryTriggerInteraction collideToTriggers = QueryTriggerInteraction.Collide;

    private Texture2D tex;

    private DetectEnvironment detector;

    private float[] pixel;

    private float maxPixel;

    private int u;
    private int v;
    private Vector2 pixelUV;

    RaycastHit hit;
```

Figure 14. HeatMapGenerate script properties.

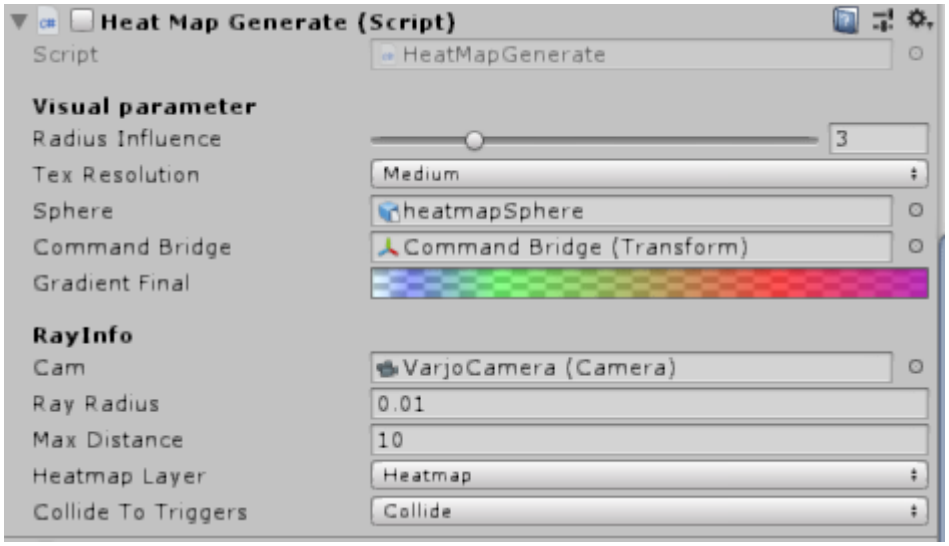


Figure 15. HeatMapGenerate script variables from Unity Inspector.

The RadiusInfluence is how big the contact point is in the sphere. TexResolution decides the details of the Texture2D created for the sphere material[13]. The higher the TexResolution is, the greater height and width of Texture2D. The sphere in the script is the invisible sphere game object. The commandBridge is the transform of the ship where the player moves around. This transform will use as an anchor for navigation. The gradient determined the color of the heatmap. It goes from blue to purple based on how long the player looked at that area. And all the ray information for navigation.

```
void Awake (){

    detector = GetComponent<DetectEnvironment>();

    switch (TexResolution){
        case Size.veryLow:
            tex = new Texture2D(32,16);
            break;
        case Size.low:
            tex = new Texture2D(64,32);
            break;
        case Size.medium:
            tex = new Texture2D(128,64);
            break;
        case Size.high:
            tex = new Texture2D(256,128);
            break;
        case Size.veryHigh:
            tex = new Texture2D(512,256);
            break;
        case Size.ultra:
            tex = new Texture2D(2048,1080);
            break;


    }

    sphere.GetComponent<MeshRenderer>().material.mainTexture = tex;

    //create pixel variable  depending on size texture
    pixel = new float[tex.width*tex.height];

}


void Start (){
    ResetValues();
}

void Update() {
    DrawListHeatmap();
}
```

Figure 16. HeatmapGenerate Unity's base function.

For the Unity default function, the Awake() function will be called first. It is called when GameObject that contains the script is initialized[14]. In the Awake() function, it gets the reference for the DetectEnvironment script to get all the data from there. After that, it initialized the Texture2D based on TexResolution and set it for sphere material texture. The last step is to initialize the pixel variable depending on the size of the texture. The next function is Start()[15], and it will be called on the frame when the script is enabled before

the Update() function is called for the first time[16]. It calls the ResetValues() function to reset all the pixel values to 0. The last one is Update() function. This function will be called every frame, and inside it is called the DrawListHeatmap() function. It means that in every frame, the heatmap will be rendered based on the data collected from the DetectEnvironment script.

```
void DrawListHeatmap()
{

    for (int x = 0; x < tex.width; x++)
    {
        for (int y = 0; y < tex.height; y++)
        {

            int i = x + tex.width * y;
            //take the highest value of the pixels
            if (pixel[i] > maxPixel)
            {
                maxPixel = pixel[i];
            }
            Color colorUpdate = gradientFinal.Evaluate(pixel[i] / maxPixel);
            tex.SetPixel(x, y, colorUpdate);
        }
    }

    //raycast in the centerof viewport
    foreach(var point in detector.hitPoints)
    {
        RaycastHit hit;

        if(Physics.Linecast(commandBridge.TransformPoint( point), cam.transform.position, out hit, heatmapLayer, collideToTriggers))
        {
            Debug.DrawLine(cam.transform.position, point, Color.green);

            Renderer rend = hit.collider.GetComponent<Renderer>();
            MeshCollider meshCollider = hit.collider as MeshCollider;

            if (rend == null || rend.sharedMaterial == null || rend.sharedMaterial.mainTexture == null || meshCollider == null)
            {
                continue;

            }
            pixelUV = hit.textureCoord;

            //coordinates in depending on height and width
            pixelUV.x *= tex.width;
            pixelUV.y *= tex.height;

            //check position of pixel
            tex.GetPixel((int)pixelUV.x, (int)pixelUV.y);

            //choose radius of the circle
            float rSquared = radiusInfluence * radiusInfluence;

            //for each pixel of texture
            for (int u = (int)pixelUV.x - (int)radiusInfluence; u < (int)pixelUV.x + (int)radiusInfluence + 1; u++)
                for (int v = (int)pixelUV.y - (int)radiusInfluence; v < (int)pixelUV.y + (int)radiusInfluence + 1; v++)

                    //create circle
                    if ((pixelUV.x - u) * (pixelUV.x - u) + (pixelUV.y - v) * (pixelUV.y - v) < rSquared)
                    {
                        //edit the value of the pixel, adding deltatime and making a gradient from the center
                        int PixCurrent = u + tex.width * v;
                        pixel[PixCurrent] += Time.deltaTime * (1f - ((pixelUV.x - u) * (pixelUV.x - u) + (pixelUV.y - v) * (pixelUV.y - v)) / radiusInfluence * 0.05f);
                    }
        }

    }
    //apply texture
    tex.Apply();
}
```

Figure 17. DrawListHeatmap function.

Let’s take a closer look at DrawListHeatmap() function. First of all, it runs a loop through all the pixel values and finds the max value. Based on the ratio of the value of the pixel and the max value, the color for that pixel will be chosen from

the given gradient. The next step is to run through every contact point detected from the DetectEnvironment script. For each point, cast a Line from that position to the player position and find a collision with the object with the given layer. In this case, the sphere will be specific to that layer. The line cast will always hit the sphere in theory because the sphere is surrounding the player. From the collision point between line cast and sphere, that point coordinate will be translated to pixel coordinate position. At that position, a circle will be created based on the given radiusInfluence. The last loop will run through every pixel of the texture and edit the value of the one inside of the created circle. The final step is to apply all the changes to the texture.

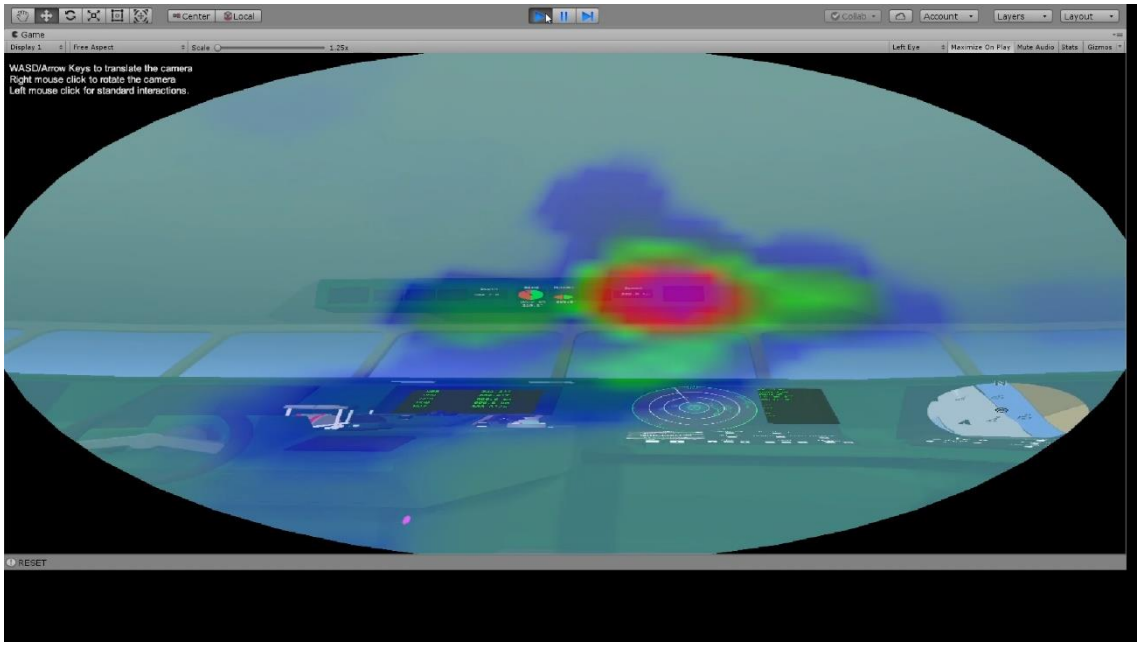


Figure 18. Heatmap outcome.

The result will somewhat look like Figure 18 in the game. The heatmap shown in-game is just for testing purposes. Because if the heatmap shows it while the player is playing, it is not practical and will distract the player from his best performance. So the heatmap will stay disabled in-game and only be active when extracting the 360 images.

5.3 Scanpath

The third method would be scanpath. The way of building this method is slightly similar to the heatmap. Nevertheless, in the end, because the application for this method is different and has more details than the heatmap, building it is also slightly complicated. The heatmap shows the intensity of eye motion in a particular area. Other than heatmap, scanpath shows how eye fixations travel in that area. As a result, the scanpath would be more complicated, and many more aspects need to be mentioned. The scanpath includes several factors such as where, how long, and the order of it.

Let’s look at the first factor, where is the player looking? In this thesis project, the author chooses game objects in the environment as the measurement. Instead of creating a fixation every time the player's eye movement change, it creates a fixation every time player change his intention on another object. By using this approach, the number of fixations will be decreased and the scanpath will be more simple and easier to analyze. As a result, the amount of data to collect to render the scanpath also decreases, and increases the performance. In order to archive the best performance for this method, the collider for the game object needs to be optimized. In addition, each game object attached a VarjoGazeTarget script.

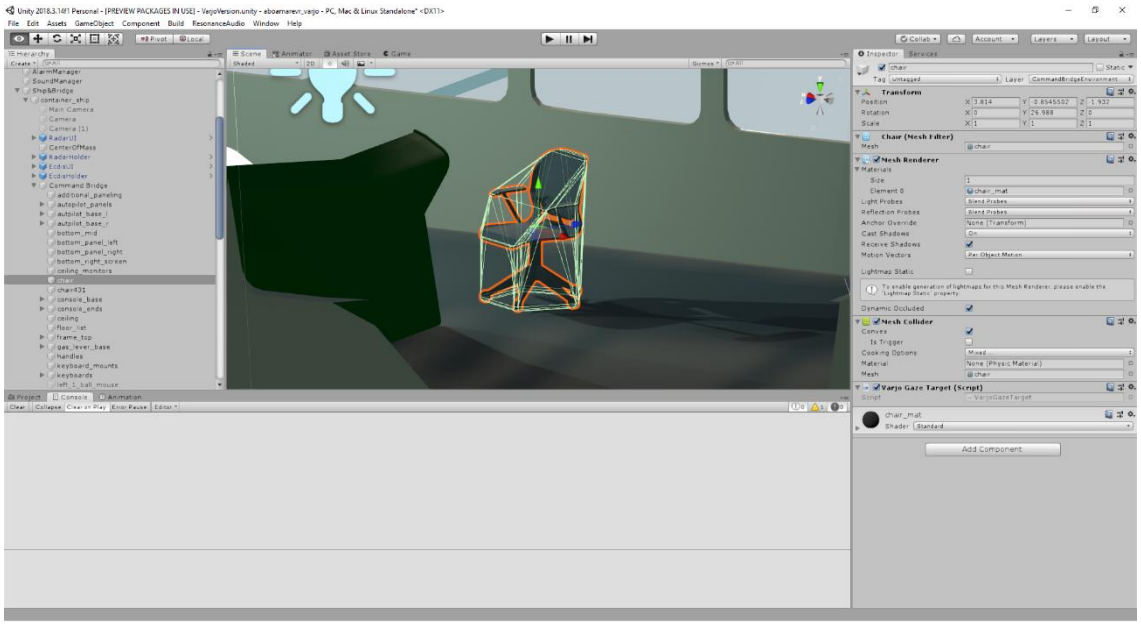


Figure 19. Object's collider.

The more accurate the collider is, the better the scanpath is. Because of that, the more collider is better. Especially the area has multiple objects overlapping each other, and the collider needs to be carefully set. For sizeable objects and one-piece objects, multiple colliders are needed. By separating the oversize object into smaller game objects, especially the object with various details, the result will be more accurate and it will also present more helpful information to the analyst.

```
void DetectGazeTargetWithVarjoGaze()
{
    if (!detecting)
        return;

    gazeData = VarjoPlugin.GetGaze();

    if (gazeData.status != VarjoPlugin.GazeStatus.INVALID)
    {
        gazeForward = new Vector3((float)gazeData.gaze.forward[0], (float)gazeData.gaze.forward[1], (float)gazeData.gaze.forward[2]);
        gazePosition = new Vector3((float)gazeData.gaze.position[0], (float)gazeData.gaze.position[1], (float)gazeData.gaze.position[2]);

        transform.position = VarjoManager.Instance.HeadTransform.position;
        transform.rotation = VarjoManager.Instance.HeadTransform.rotation;

        // Transform gaze direction and origin from HMD space to world space
        sphereCastDirection = transform.TransformVector(gazeForward);
        sphereCastOrigin = transform.TransformPoint(gazePosition);

        if (Physics.SphereCast(sphereCastOrigin, rayRadius, sphereCastDirection, out gazeRayHit, maxDistance, environmentLayer.value, collideToTriggers))
        {
            VarjoGazeTarget hitTarget = gazeRayHit.collider.GetComponent<VarjoGazeTarget>();
            if (hitTarget == null)
                return;

            if (currentTarget == null)
            {
                timer = 0;
                currentTarget = hitTarget;
                contactPoint = commandBridge.InverseTransformPoint(gazeRayHit.point);
            }
            // Add gaze target when player look to another gaze target
            if (hitTarget != currentTarget)
            {
                Add(currentTarget, timer, contactPoint);
                timer = 0;
                currentTarget = hitTarget;
                contactPoint = commandBridge.InverseTransformPoint(gazeRayHit.point);
            }
        }
    }
}
```

Figure 20. DetectGazeTargetWithVarjoGaze function.

The navigation method was nearly the same as with the heatmap. It also gets the gaze information and casts a SphereCast into the environment to find the object. After finding the environment object, it checks for the VarjoGazeTarget script. If the object does not have VarjoGazeTarget script attached to it, the system will skip. If the CurrentTarget is null, which is the first target player looking at, then it will assign the detected target into the CurrentTarget variable. At the same time, it starts the timer to count how long the player looks at that object and assigns the contact point to the ContactPoint variable. In another case, where the detected target is not the CurrentTarget, it will add the current target information into a list of ScanpathPoint(Figure21). Then, reset the timer and assign a new value to the CurrentTarget and ContactPoint variables.

```
public class ScanpathPoint
{
    public int id;
    public string objectName;
    public VarjoGazeTarget targetObject;
    public Vector3 contactPoint;
    public float duration;
}
```

Figure 21. ScanpathPoint class.

```
void Timer()
{
    timer += Time.deltaTime;
}

void Add(VarjoGazeTarget target, float duration, Vector3 contactPoint)
{
    ScanpathPoint currentPoint = new ScanpathPoint();
    currentPoint.objectName = target.gameObject.name;
    currentPoint.targetObject = target;
    currentPoint.duration = duration;
    currentPoint.contactPoint = contactPoint;
    currentPoint.id = scanpathPointsList.Count;
    scanpathPointsList.Add(currentPoint);
    if(duration> maxDuration)
    {
        maxDuration = duration;
    }
}
// Add list of scanpath point to a sequence to compared with other player
public string CollectScanpathData()
{
    string scanpathStringSequence = "";

    foreach(var point in scanpathPointsList)
    {
        scanpathStringSequence += point.objectName + "-";
    }
    return scanpathStringSequence;
}

public void StopDetection()
{
    detecting = false;
}
```

Figure 22. DetectEnvironment subfunction.

The Add() function take three parameters including VarjoGazeTarget target, float duration and Vector3 ContactPoint. This three-parameter matched with timer, CurrentTarget, and ContactPoint variables mentioned earlier. Inside the Add() function, it creates a ScanpathPoint object and passes the value into it. Furthermore, the ScanpathPoint's objectName will be passed the target object name into it. Each ScanpathPoint will have an identical ID for itself. The final step is to add the ScanpathPoint into a list to store the data collected and assign the new MaxDuration value if the duration satisfies the condition. The list will be used for rendering the fixation of the scanpath. Furthermore, it is also used to extract scanpath sequence strings. The sequence string is the combination of object's names in order of scanpath, and the CollectScanpathData() function is the one for that. The function will be called at the end of the scanpath and return the sequence string of the whole list at once.

To handle the scanpath rendering and animation, the ScanpathManager script is in use. All the parameters with SerializeField will be initialized from Unity’s inspector.

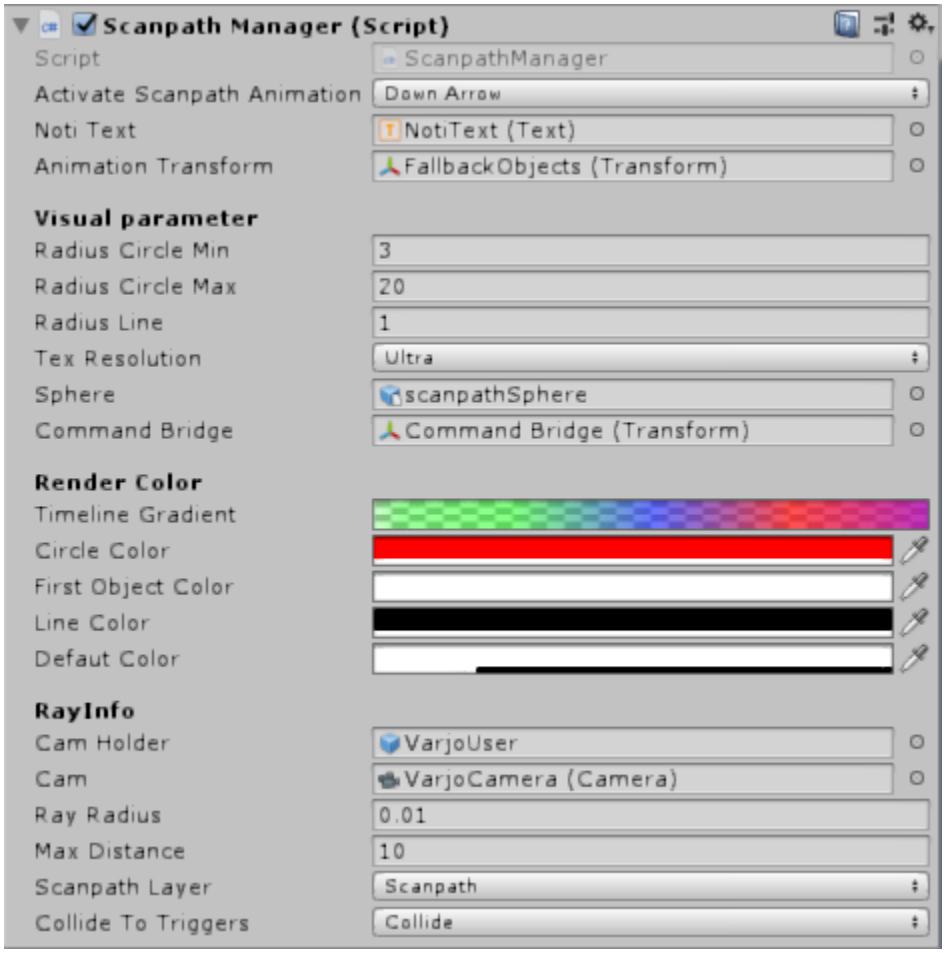


Figure 23. ScanpathManager script variables from Unity Inspector.

Overall, all the parameters used in the HeatmapGenerator script are also being used here. TexResolution, Gradient, Sphere, CommandBridge transform, and RayInfo remain the same uses. On the other hand, the rest of the parameters are used for scanpath own purposes. Let's take a look at each of them. The first one is the ActivationScanpathAnimation key bind. By pressing this key bind, the game will enter into the scanpath animation mode. The NotiText will be the notification of it. When entering the animation mode, the player will be moved to the AnimationTransform position to have a better view of the whole room. The position will be in the middle of the command bridge. The next is visual parameters. For scanpath rendering, all the fixations will be represented as circles and joined together by a line. All the circles and line radius will be

initialized from Unity's inspector like Figure 23. About the color, all the colors will be initialized from the Unity's inspector. Circle color and line color will describe the fixations and the lines between colors. The timeline gradient will show the order of the scanpath by colors. The fixations go from white to purple, depending on the order of it. The warmer the color is, the later the fixations.

All of the variables that are initialized from the Unity's inspector will be used inside the ScanpathManager script. Let's take a look inside it. The Awake() and Start() function remains the same with HeatmapGenerate. The Update() function is different in this script. Instead of drawing a heatmap in every frame, it will check if the player is entering the Animation mode or not. If the condition is satisfied, it will call the function to render the scanpath.

```
private void Update()
{
    if (Input.GetKeyDown(activateScanpathAnimationKey))
    {
        ResetValues();
        enableAnimation = true;
        StartCoroutine(ScanpathAnimation(5f));
        scanpathDataLog.Log(detector.CollectScanpathData());
    }
}
private void LateUpdate()
{
    if (enableAnimation)
    {
        cam.transform.localPosition = animationTransform.localPosition;
    }
}
```

Figure 24. ScanpathManager Unity base function.

First, it will ResetValues() just like HeatmapGenerate does. Then it changes the boolean variable EnableAnimation to true. Doing this will freeze the player's

position in the LateUpdate() function. Then it will start the Coroutine of ScanpathAnimation and start logging data from the ScanpathDataLog script.

```
IEnumerator ScanpathAnimation(float speed)
{
    if (!enableAnimation)
        yield break;

    detector.StopDetection();
    float currentRadius = 0f;
    Vector3 previousContactPoint = Vector3.zero;
    foreach (var point in detector.scanpathPointsList)
    {
        RaycastHit hit;
        if (Physics.Linecast(commandBridge.TransformPoint(point.contactPoint), cam.transform.position, out hit, scanpathLayer, collideToTriggers))
        {

            Renderer rend = hit.collider.GetComponent<Renderer>();
            MeshCollider meshCollider = hit.collider as MeshCollider;

            if (rend == null || rend.sharedMaterial == null || rend.sharedMaterial.mainTexture == null || meshCollider == null)
            {
                continue;
            }

            pixelUV = hit.textureCoord;

            //coordinates in depending on height and width
            pixelUV.x *= tex.width;
            pixelUV.y *= tex.height;

            //check position of pixel
            tex.GetPixel((int)pixelUV.x, (int)pixelUV.y);

            float radiusRatio = point.duration / detector.maxDuration;
            currentRadius = 0;

            float radiusCircle = radiusRatio * (radiusCircleMax - radiusCircleMin) + radiusCircleMin;

            if (previousContactPoint != Vector3.zero)
            {
                Vector3 lineVector = commandBridge.TransformVector(point.contactPoint - previousContactPoint);

                float distance = Vector3.Distance(point.contactPoint, previousContactPoint);

                for (float i = 0; i < distance; i += 0.001f * distance)
                {
                    Vector3 p = i * Vector3.Normalize(lineVector) + commandBridge.TransformPoint(previousContactPoint);

                    if (Physics.Linecast(p, cam.transform.position, out hit, scanpathLayer, collideToTriggers))
                    {

                        rend = hit.collider.GetComponent<Renderer>();
                        meshCollider = hit.collider as MeshCollider;

                        if (rend == null || rend.sharedMaterial == null || rend.sharedMaterial.mainTexture == null || meshCollider == null)
                        {
                            continue;
                        }

                        pixelUV = hit.textureCoord;
```

Figure 25. ScanpathAnimation Coroutine.

```
                    pixelUV = hit.textureCoord;

                    //coordinates in depending on height and width
                    pixelUV.x *= tex.width;
                    pixelUV.y *= tex.height;

                    //check position of pixel
                    tex.GetPixel((int)pixelUV.x, (int)pixelUV.y);

                    //choose radius of the circle
                    float rSquaredLine = radiusLine * radiusLine;

                    //for each pixel of texture

                    for (int u = (int)pixelUV.x - (int)radiusLine; u < (int)pixelUV.x + (int)radiusLine + 1; u++)
                        for (int v = (int)pixelUV.y - (int)radiusLine; v < (int)pixelUV.y + (int)radiusLine + 1; v++)

                            //create circle
                            if ((pixelUV.x - u) * (pixelUV.x - u) + (pixelUV.y - v) * (pixelUV.y - v) < rSquaredLine)
                            {
                                int PixCurrent = u + tex.width * v;
                                tex.SetPixel(u, v, lineColor);
                            }

                    }

                }
                tex.Apply();
            }

            while (currentRadius < radiusCircle)
            {
                float rSquared = currentRadius * currentRadius;
                for (int u = (int)pixelUV.x - (int)currentRadius; u < (int)pixelUV.x + (int)currentRadius + 1; u++)
                    for (int v = (int)pixelUV.y - (int)currentRadius; v < (int)pixelUV.y + (int)currentRadius + 1; v++)

                        //create circle
                        if ((pixelUV.x - u) * (pixelUV.x - u) + (pixelUV.y - v) * (pixelUV.y - v) < rSquared)
                        {
                            int PixCurrent = u + tex.width * v;

                            if (point.id == 0)
                                tex.SetPixel(u, v, firstObjectColor);
                            else
                            {
                                float gradientValue = (float)point.id / (float)detector.scanpathPointsList.Count;
                                tex.SetPixel(u, v, timelineGradient.Evaluate(gradientValue));
                            }

                        }
                tex.Apply();
                currentRadius += Time.deltaTime * speed;
                yield return new WaitForEndOfFrame();
            }
            previousContactPoint = point.contactPoint;
        }
    }
}
```

Figure 26. ScanpathAnimation Coroutine.

The ScanpathAnimation() handles the scanpath animation in the Animation mode. Bypassing a speed value when calling, it can decide the speed of the animation. First, it checks if the player is in Animation mode. If not, it will break out of the function. Continue with stopping collecting all the data from DetectEnvironment. Following up, it creates two variables for storing the current radius of the circle and the previous contact point. The next step remains the same with HeatmapGenerate to find the contact point on the Sphere surface.

Furthermore, from that contact point found from the collision, it will be translated to pixel coordinate. Since now the pixel coordinate has been navigated, the next step is rendering the fixation on that position. There are two properties for fixation radius that have been initialized from the Unity's inspector: RadiusCircleMax and RadiusCircleMin. The radius of the fixation will scale from that min and max values based on the ratio of the fixation duration to the maximum duration. This scale is to balance the circle radius and the environment. It would be hard for the analyst to evaluate the scanpath if the fixation size is overflowing the main object and overriding other objects. Moreover, on the other hand, it would also be hard to evaluate if the size of the fixation is too small. So by using the maximum and minimum values for the fixations circle, the analyst can customize the scanpath based on their opinions and make it the most convenient for them to research their work. So for each ScanpathPoint collected from DetectEnvironment, calculated RadiusRatio based on the point duration and the maximum duration. Furthermore, from that ratio, combined with two given radius variables, it calculated the fixation radius. To animate the circle, a while loop has been used. The condition for the while loop is CurrentRadius < RadiusCircle. The CurrentRadius variable has been assigned as 0. At the end of the while loop, it increased the value based on Unity delta time and the speed parameter. Also, it return a WaitForEndOfFrame(). The Coroutine principle is to pause the execution, and the resume is based on the given condition[17]. In this case, the condition is the return type of it which is the end of a frame. So the loop will pause until the end of the frame and then continue it in the next frame. In conclusion, the circle will be animated bigger and bigger from 0 to the given radius over every frame. That is what it does to animate the fixation.

The next step is to render the line. There is a PreviousContactPoint variable reserved for storing the position of the previous contact point in the loop. At first, it will be assigned as a (0,0,0) vector. For each ScanpathPoint in the for loop, it will check if the PreviousContactPoint is available. If it equals vector zero, which

means it is the first contact point, it will skip the step. If the PreviousContactPoint is available, first of all, draw a line between 2 contact points. Furthermore, after getting the line vector between the two contact points, it will projectile the whole vector to the Sphere surface with the same method as earlier used. For line rendering, it used the line radius variable that has been initialized from the Inspector. The last step is to apply the texture to the Sphere and assign the contact point to the PreviousContactPoint variable. When the animation is finished, the final result will be presented on screen to the observer. At the same time, the sequence of all fixations will also be logged to a CSV file in the ScanpathDataLog script. For the last step, the analyst can extract the final result as a 360 image.

Besides the animation, there is another option for this scanpath method. The animation approach is helpful for the analyst when it shows the progress of the player's eyes motion. However, at the same time, it is considerably time-consuming even when it has been sped up significantly. So the other option is also built-in as a function for the ScanpathManager script. The RenderScanpath() function will render the scanpath all at once and extract a 360 image as the final result. The way it works is the same as the heatmap method. Instead of using Coroutine to render it over time, the function just runs through the for loop of the contact point and renders the fixations based on the calculated radius.

```
public void Log(string sequence)
{
    DateTime now = DateTime.Now;

    string fileName = string.Format("ScanpathDataLog-{0}-{1:00}-{2:00}-{3:00}-{4:00}", now.Year, now.Month, now.Day, now.Hour, now.Minute);
    string logPath = Application.dataPath + "/Logs/";
    Directory.CreateDirectory(logPath);

    string path = logPath + fileName + ".csv";
    writer = new StreamWriter(path);

    writer.WriteLine(sequence);
    writer.Flush();
    writer.Close();
    Debug.Log("Scanpath Logged");
}
```

Figure 27. Log function.

After the scanpath has been rendered, the scanpath sequence will be logged into a CSV file. The ScanpathDataLog script will handle the logging. The Log() function will get the sequence string as the parameter and write it into the CSV file. According to the analyst, comparing scanpath sequences will assist in the scoring system he is trying to archive. By using the Needleman-Wunsch algorithm[18], the player sequence will be compared to a sample sequence provided by an experienced sea captain. The score will be given based on the match, mismatch, and gaps between the two sequences.

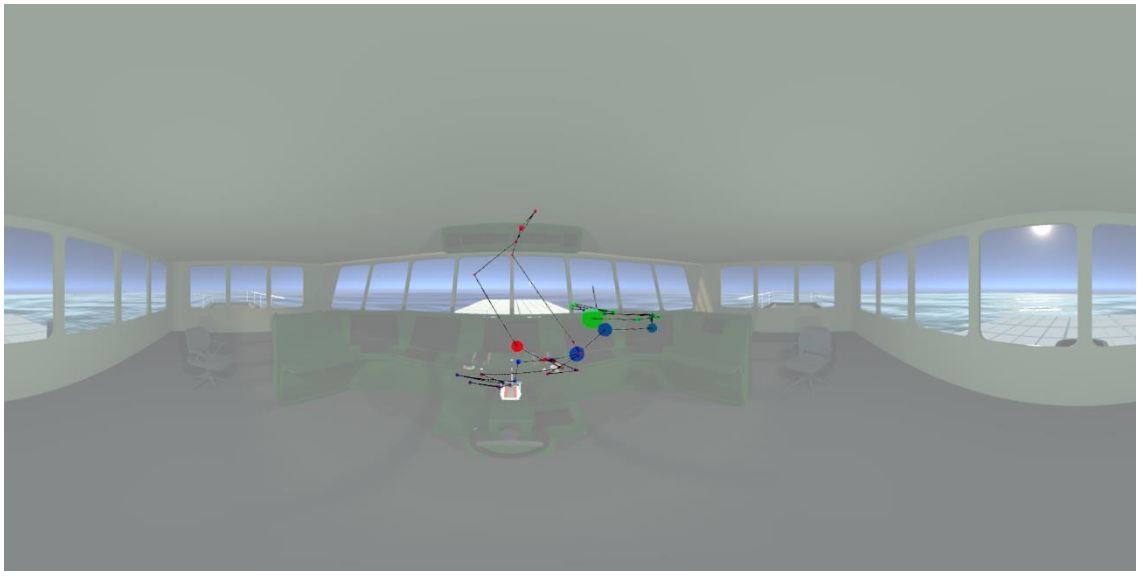


Figure 28. Scanpath outcome.

5.6 360 image

The result of the heatmap, as well as scanpath, will be rendered on top of a sphere inside the game environment. So it gives a hard time for an analyst to get access to that data. At the same time, depending on the player's position inside the environment, the result will be rendered differently because of the difference between points of view. So for the standard output of results coming out of the method, it needs to have similarities in different respects. The position inside of the VR environment should be the same as well as the point of view. Furthermore, it gives an advantage for analysts since the result can be indicated the differences at first eyes sight.

The standard output that the author chooses to extract is 360 images. The reason is simple. With 360 images, it can efficiently present the result in the 2D form either on a screen monitor as well on paper. Also, it can present fully the VR environment, including a heatmap or scanpath on top of it. The position author choose for the 360 images is from the middle of the VR environment, so the environment will be presented as detailed as possible. The shot will be taken from the player's perspective, where the sphere is surrounded. Because of this setup, the camera will be inside the sphere. From the camera's point of view, the sphere will render the heat map and scanpath in between the camera and the environment.

The code for extracting the 360 images from Unity is held in TakeScreenshotEquirectangular script.

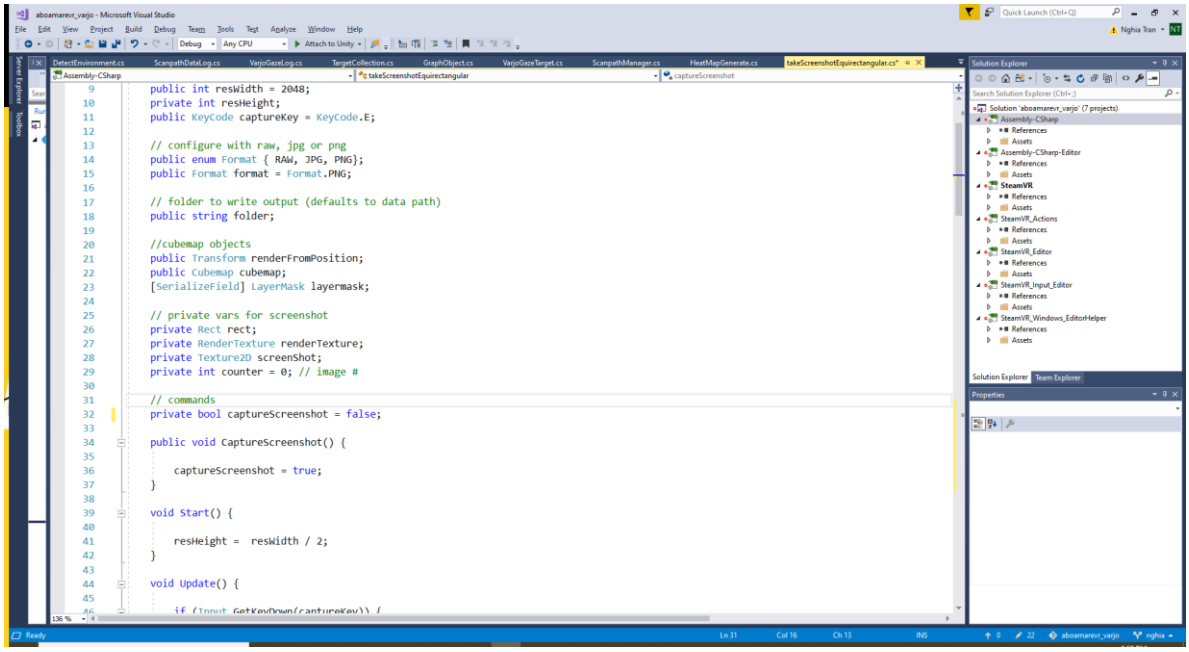


Figure 29. TakeScreenshotEquirectangular script properties.

The parameters are variables for the image. It included resolution, format, position, layer mask as well as a cube map. How the cube map work is that the 360 images will be combined with six different images. These images getting shot from the camera will be rendered in this cube map[19].

There are also private variables for the 360 image, including a 2d Rectangle (rect), a render texture, and a texture 2D.

```
private void CaptureScreenShot360()
{
        captureScreenshot = false;
        GameObject go = new GameObject("CubemapCamera");
        go.AddComponent<Camera>();
        go.transform.position = renderFromPosition.position;
        go.GetComponent<Camera>().nearClipPlane = 0.01f;
        go.GetComponent<Camera>().cullingMask = layermask;
        go.GetComponent<Camera>().RenderToCubemap(cubemap);
        DestroyImmediate(go);

        if (renderTexture == null)
        {
            rect = new Rect(0, 0, resWidth, resHeight);
            renderTexture = new RenderTexture(resWidth, resHeight, 24);
            screenShot = new Texture2D(resWidth, resHeight, TextureFormat.RGB24, false);
        }

        Camera camera = this.GetComponent<Camera>();
        camera.enabled = true;
        camera.cullingMask = layermask;
        camera.targetTexture = renderTexture;
        camera.Render();

        RenderTexture.active = renderTexture;
        screenShot.ReadPixels(rect, 0, 0);

        camera.targetTexture = null;
        RenderTexture.active = null;

        string filename = uniqueFilename((int)rect.width, (int)rect.height);

        byte[] fileHeader = null;
        byte[] fileData = null;
        if (format == Format.RAW)
        {
            fileData = screenShot.GetRawTextureData();
        }
        else if (format == Format.PNG)
        {
            fileData = screenShot.EncodeToPNG();
        }
        else
        {
            fileData = screenShot.EncodeToJPG();
        }

        new System.Threading.Thread(() => {
            var f = System.IO.File.Create(filename);
            if (fileHeader != null) f.Write(fileHeader, 0, fileHeader.Length);
            f.Write(fileData, 0, fileData.Length);
            f.Close();
            Debug.Log(string.Format("Wrote screenshot {0} of size {1}", filename, fileData.Length));
        }).Start();

        Destroy(renderTexture);
        renderTexture = null;
        screenShot = null;

}
```

Figure 30. CaptureScreenshot360 function.

To handle rendering the 360 images, the CaptureScreenShot360() method is triggered when the user presses the capture key that is preset from Inspector. This method creates a temporary camera for rendering by creating a new object and adding a Camera component[20]. And then, based on how users want to

render the 360 images, it can be customized for different uses. The author uses a layer mask of everything so it can render both the environment as well as the heatmap and scanpath. By turning off a layer mask bit, the user can choose what will be rendered in the 360 images. After setting the necessary parameters for the camera component to work, it will render the camera view into a cube map and destroy that temporary camera.

If the screenshot is needed, it will also create new screenshot objects. The next step is to get the main camera and manually render the scene into a render texture. After that, create a new unique file name and extract it into different forms. Users can choose raw, png, and jpg format for the image.

7 Results

During the developing time of the thesis project, testing was simplified as much as it can be. The technology that has been built is for MarISOT main project interest. Project has given was a student project and it still exists a significant amount of bugs that affect the development process. The approaching way to solve the problems also has to work around those bugs. Because of that reason, the tools were built isolated from the rest of the demo project. So the testing was simplified to test only the visualization tools but not how they work with the whole project. The application of this tool and further development as well as optimizing it with the main project will be handled by GameLab staff in the different phases of the project.

The visualization tool is tested by the author and considered as complete.

The graph was successfully built and it runs and updates the data in real-time with the game. The scanpath animation is also successful and well presented. The demo video is attached as an URL link.

Video Url:

https://vimeo.com/674188475

https://vimeo.com/674189048

The heatmap and scanpath result as a 360 image is also successfully presented in Figure 18 and Figure 28.

As soon as the tool was completed, the author contacted the analyst to get more feedback on the tool. After some testing and analysis, the feedback from the analyst was positive.

8 Discussion

This thesis study and development support an analyst's master thesis. His topic is to study a grading system for the training episode. To archive that, he isolated and researched every factor in eye-tracking data. When an in-game event happens, having a tool to inspect how the data change during that time is helpful for the research. Based on his description and requirement, the author established this tool and delivered it to him for his further research. The result got positive feedback from the analyst, and it provided what he needed for this research. With some of the test cases, overall, he can certainly see the player's behavior. The result captured the most exceptional data change moment based on in-game events. Factors like pupil size and focus stability in those in-game events can be decided in the final grade. The heatmap result shows the majority of time players spend in a particular area and object. It also demonstrates how to concentrate the player on important spaces like the controller board. The analyst was also happy with the scanpath sequence in-game object. With the sequence result, he can develop a draft grading system using the Needleman-Wunsch algorithm. In conclusion, the tool met the analyst's expectations and became a valuable resource for his further study. On the other hand, some feedback from the analyst and suggestion from the author for continuous development is worth mentioning. For the demo purpose, the performance, and testing is limited. For further development, the number of data should be decreased. It should only be focusing on some data that analysts think is worth inspecting. Without a doubt, the chosen data will be the result of the analyst study. By decreasing the number of data, the performance also increases. Also, by increasing the number and accuracy of the colliders, performance will also be increased significantly. But for demo purposes, analyst feedback about the performance is acceptable.

9 Conclusion

The objective of this thesis was to develop a tool to visualize eye-tracking data. To archive this objective, the author studied previous studies from existing research about the same topic. Because the previous studies were not fit to the thesis purpose, a new navigation method was built and fill the gap. The newly developed navigation method is built as the core and from there other methods can be built on top of it.

The tool was developed successfully and got well-received by the eye-tracking specialist. The outcome of the developed tool fulfills the visualization requirements as requested by the specialist. The requirements include heatmap, scanpath, object sequence, and finally, isolate and visualize eye-tracking data. This thesis practical outcome also contributes to previous research about eye-tracking visualization by providing a novel visualization tool for eye-tracking data. The research on navigation in the VR world not only applies to this particular tool development but could also be used in many other aspects of VR game development, such as player's position and navigation, camera angle, etc.

References

[1] Video game industry statistics, trends and data in 2022. Jan 18, 2022

https://www.wepc.com/news/video-game-statistics/

[2] Micheal Mamerow: Gaming industry vs Other Entertainment industries(2022)

https://raiseyourskillz.com/gaming-industry-vs-other-entertainment-industries-2021/

[3] Mika Luimula, Evangelos Markopoulos, Johanna K Kaakinen, Panagiotis Markopoulos, Niko Laivuori, Werner Ravyse: Eye Tracking in Maritime Immersive Safe Oceans Technology. September 2020

11th IEEE International Conference on Cognitive Infocommunications. CogInfoCom 2020At: Online on MaxWhere 3D September 2020 DOI: 10.1109/CogInfoCom50765.2020.9237854

[4] Wikipedia: Heat map. 4th May 2022

https://en.wikipedia.org/wiki/Heat_map

[5] Scanpaths: Definitions, theory and applications

https://www.cis.rit.edu/pelz/scanpaths/scanpaths.htm

[6] Tanja Blascheck, Kuno Kurzhals, Michael Reascheke, Micheal Burch, Daniel Weiskopf, Thomas Ertl: Visualization of Eye Tracking Data: A Taxonomy and Survey: Visualization of Eye Tracking Data

Computer Graphics Forum 36. February 2017 DOI:10.1111/cgf.13079

[7] Martin Mirchev, Job van de Ven, Jan Willem van Ringelesteijn, Alex Perez-Lucerga Garrido, Lucas Gether Ronning, Isa Dantuma, and Michael Burch : Visualization of Eye Tracking Data

Education in Visualization. May 2020 DOI: 10.13140/RG.2.2.19353.65121

[8] Varjo Document: Unity. May 2022

https://developer.varjo.com/docs/v2.1.0/unity/unity

[9] Unity Document: Sphere cast. May 2022

https://docs.unity3d.com/ScriptReference/Physics.SphereCast.html

[10] Unity document: Line renderer. May 2022

https://docs.unity3d.com/ScriptReference/LineRenderer.html

[11] Unity document: Canvas. May 2022

https://docs.unity3d.com/2020.1/Documentation/Manual/UICanvas.html

[12] Unity Document: Material. May 2022

https://docs.unity3d.com/ScriptReference/Material.html

[13] Unity Document: Texture. May 2022

https://docs.unity3d.com/Manual/Textures.html

[14] Unity Document: Awake. May 2022

https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html

[15] Unity Document: Start. May 2022

https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html

[16] Unity Document: Update. May 2022

https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

[17] Unity Document: Coroutines. May 2022

https://docs.unity3d.com/Manual/Coroutines.html

[18] Vladimir Liki´c: The Needleman-Wunsch algorithm for sequence alignment. May 2022

https://www.cs.sjsu.edu/~aid/cs152/NeedlemanWunsch.pdf

[19] Unity Document: Cubemaps. May 2022

https://docs.unity3d.com/Manual/class-Cubemap.html

[20] Unity Document: Camera. May 2022

https://docs.unity3d.com/ScriptReference/Camera.html