FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor’s thesis

Title:	Tracking of user’s view in virtual reality systems
Student:	Richard Kvasnica
Supervisor:	Ing. Jan Buriánek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Computer Graphics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Eye-tracking is a feature of many modern virtual reality systems. It enables determination of user's gaze by combining head rotation and eye movement inside a 3D virtual environment. In VR, this tracking has applications in graphics performance optimization and is used for "view frustum culling". The aim of this work is to utilize this feature for user behaviour analysis in a predefined 3D scene in virtual reality.
1) Research available eye-tracking systems and explore current gaze tracking solutions.
2) Describe the architecture and principles of the found eye-tracking methods. Mainly focus on those implemented in virtual reality systems.
3) Choose a VR system with eye-tracking capabilities accessible to you, and develop an app prototype that collects gaze data in a virtual 3D scene.
4) Use the app to obtain data from different users. Attempt to describe and visualize the resultant data.
5) Submit your work with images and commented source code attached.

Electronically approved by Ing. Radek Richtr, Ph.D. on 11 February 2022 in Prague.

Bachelor's thesis

TRACKING OF USER'S VIEW IN VIRTUAL REALITY SYSTEMS

Richard Kvasnica

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jan Buriánek
June 23, 2022

Citation of this thesis: Kvasnica Richard. *Tracking of user's view in virtual reality systems.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

Acknowledgments **viii**

Declaration **ix**

Abstrakt **x**

Acronyms **xii**

1 Introduction **1**

1.1 Goals 2

2 Eye-tracking technology **3**

2.1 Existing techniques 3

2.1.1 Electro-oculography 3

2.1.2 Scleral contact lens 4

2.1.3 Video-based methods 4

2.2 Eye detection 6

2.2.1 Shape 7

2.2.2 Appearance 7

2.2.3 Dark and Bright pupil 8

2.2.4 Purkinje images 8

2.3 Gaze estimation 9

2.3.1 2D regression 9

2.3.2 3D model 9

3 Analysis **11**

3.1 Gaze tracking 11

3.1.1 Fundamentals 11

3.1.2 Gaze data 12

3.1.3 Visualisations 14

3.2 Current hardware 15

3.2.1 Tobii 15

3.2.2 Integrations in VR HMDs 17

3.2.3 OpenXR standard 18

3.2.4 Varjo 18

3.2.5 Vrgineers XTAL 20

3.2.6 Future possibility 22

3.3 Existing solutions 23

3.3.1 Cognitive3D 23

3.4 Unreal Engine capabilities 26

3.4.1 Blueprint Visual Scripting 27

3.4.2 Tools and Classes 27

3.4.3 OpenXR 28

3.4.4 UV mapping 28

3.4.5 Mesh painting . . . 29
3.4.6 UV painting method . . . 29
3.4.7 UV unwrap method . . . 30
3.4.8 Runtime texture loading . . . 32

4 Design **35**
4.1 Requirements . . . 35
4.1.1 Functional requirements . . . 35
4.1.2 Non-functional requirements . . . 35
4.2 Prototype application . . . 36
4.2.1 Selected technologies . . . 36
4.2.2 Form . . . 36
4.2.3 Modification . . . 37
4.3 Functionality . . . 37
4.3.1 Actions . . . 38

5 Implementation **41**
5.1 Materials . . . 41
5.1.1 LoadMaterial and UVRenderBackground . . . 41
5.1.2 UnwrapBrush . . . 42
5.1.3 HeatMaterial . . . 42
5.1.4 TrailCompareMaterial . . . 44
5.2 Blueprints . . . 44
5.2.1 EyeTrackedStaticMeshComponent . . . 45
5.2.2 HeatmapProducer . . . 48
5.2.3 GazeEmitter . . . 50
5.3 Python . . . 52
5.3.1 EXR image merger script . . . 52

6 Experiment **53**
6.1 Scene . . . 53
6.1.1 Czech National Bank Visitor Centre . . . 54
6.2 Scene modification . . . 55
6.2.1 Joined objects . . . 56
6.2.2 Overlapping UVs . . . 56
6.2.3 Material Baking . . . 57
6.2.4 Origin of models . . . 58
6.2.5 Model export . . . 58
6.3 Unreal project . . . 58
6.3.1 Setting up the scene . . . 58
6.3.2 Pawns . . . 60
6.3.3 Controls . . . 63
6.4 XTAL . . . 64
6.4.1 Setup . . . 64
6.4.2 Troubleshooting . . . 64
6.5 Data collecion . . . 66
6.5.1 Test scenario . . . 66
6.5.2 Measurement process . . . 66
6.6 Resultant data . . . 66

7 Conclusion **73**

A Unreal Engine 5 migration **75**

B ET Component DMI initialisation **77**

C Baked texture difference **79**

Contents of the enclosed media **87**

List of Figures

2.1 Examples of intrusive eye tracking devices. 4
2.2 Typical video-oculography setup using method of corneal reflection. 5
2.3 Components of eye tracking process. 6

3.1 Detailed scheme of gaze data 12
3.2 Example of two different heatmap approaches. 14
3.3 Description of Tobii Glasses 3. 16
3.4 Interconnection of ET hardware with SDK. 16
3.5 Schema of OpenXR cross-platform access to different XR devices. 18
3.6 Varjo VR-3 virtual veality headset. 19
3.7 Mixed reality laboratory for religious studies in Olomouc Palacký University. . . 20
3.8 VRGineers XTAL 3 VR headset. 21
3.9 3D CAVE laboratory at CTU FME. 22
3.10 VRGineers fighter jet cockpit simulator. 22
3.11 Sequence of steps and their completion in Cognitive3D. 25
3.12 Setup of a step in Cognitive3D that activates at gaze. 26
3.13 Preview of 2D brush Unreal Material. 29
3.14 Difference between applied unwrap material. 30
3.15 Material Expressions used to unwrap mesh to world space. 31
3.16 3D Brush defined in Material Expressions. 32
3.17 Process of importing texture on Render Target. 33
3.18 Unreal Material used for loading. 33

4.1 Activity diagram of a possible ET experiment collecting heatmaps. 39

5.1 Contents of EyeTrackingUtilities Material folder. 41
5.2 Unwrap 3D brush material correction with angular conditions. 43
5.3 HeatMaterial node definition. 43
5.4 TrailCompareMaterial node definition. 44
5.5 Contents of EyeTrackingUtilities Blueprint folder. 44
5.6 AttachRenderer function in EyeTrackedStaticMeshComponent. 45
5.7 EyeTrackedStaticMeshComponent render target initialisation. 45
5.8 Loading heatmap textures to render targets. 46
5.9 Export gaze and camera heatmap textures as EXR image format. 47
5.10 Make filepath for the exported gaze and camera heatmap textures. 47
5.11 HeatmapProducer capture component with background plane in viewport. . . . 48
5.12 UnwrapUVCapture Component settings. 48
5.13 HeatmapProducer ApplyHeatToComponent function. 49
5.14 HeatProducer RenderSplatOntoComponent function. 50
5.15 GazeEmitter Event BeginPlay. 51
5.16 GazeEmitter get a complex world hit on a single component. 51

6.1 NC_ET Level of CNB_NC_ET Unreal project. 53

6.2 AV MEDIA SYSTEMS, a.s. internal prototype Blender model of CNB Visitor Centre. 54
6.3 Visitor Centre of Czech National Bank during construction. 54
6.4 Multiple meshes joined into one model. 55
6.5 An overlapping UV map on one information panel. 55
6.6 Blender edit mode separate selection and mark seam operations. 56
6.7 Preparation of the information panel material nodes for baking into a new texture. 57
6.8 Blender cycles settings for baking simple diffuse colour onto an ImageTexture. . . 57
6.9 Actors containing scene objects. 58
6.10 Exhibit defined as logical scene entity. 59
6.11 VRCollector movement definition. 60
6.12 VRCollector set experiment start and duration. 61
6.13 VRCollector Tick Event sending rays from virtual camera. 61
6.14 SceneObserever Event Graph. 62
6.15 Control scheme of the unreal project with experiment. 63
6.16 Vrgineers VR Tool runtime eye settings. [73] 64
6.17 UEEditor crash log after OpenXR VR Preview. 65
6.18 Heatmap texture variants of Panel11 object. 68
6.19 Resultant heatmap of Panel11. 68
6.20 Heatmap texture variants of Panel14 object. 69
6.21 Resultant heatmap of Panel14. 69
6.22 Heatmap texture variants of Panel17 object. 70
6.23 Resultant heatmap of Panel17. 70
6.24 Heatmap texture variants of Panel2 object. 71
6.25 Resultant heatmap of Panel2. 71

List of Tables

3.1 Specification of the latest VR headsets with Tobii ET integration. 17
3.2 Cognitive3D gaze metrics of dynamic objects. 24
3.3 Cognitive3D metrics used for objectives and its steps. 25

First of all, I would like to express my gratitude to my supervisor who allowed me to work on this topic and offered me a lot of useful and factual advice. Next, I would like to thank my friends for their moral and mental encouragement, but, most importantly, I want to express my deepest gratitude to my family. For their constant support, love, and the nerves of steel throughout my studies.

Declaration

In Prague on June 23, 2022 ....

Abstrakt

Obsahem této práce je průzkum možností sledování očí uživatele za pomoci brýlí pro virtuální realitu, analýza existujících řešení, tvorba prototypu aplikace zabývající se sběrem a vizualizací pohledových dat, a jeho použití pro výrobu testovací scény experimentu, která slouží ke sběru těchto dat od vícero uživatelů. Z nasbíraných dat se vytvoří výsledná data, která se analyzují. Prototyp aplikace je koncipován jako zásuvný modul pro *Unreal Engine 4* (UE4). Pro vizualizaci a sběr pohledových dat je použit způsob vyrábění teplotních map pro každý objekt ve scéně zvlášť. Popisuje se návrh modulu, jeho implementace a jeho použití na výrobu experimentu testující průchod novým Návštěvnickým centrem České Národní Banky.Scéna je v Blenderu a práce popisuje její nutné úpravy potřebné pro export do UE4 k zajištění korektní funkčnosti experimentu. Výsledkem této práce je funkční prototyp zásuvného modulu pro UE4 schopný záznamu pohledových dat skrze vyrábění teplotních map v reálném čase, který není závislý na konkrétním hardwaru pro sledování očí. Dále nabízí popis použití brýlí XTAL, které nebylo možné zprovoznit ke sběru pohledových dat z důvodu nestability jejich OpenXR běhového prostředí. Jiná alternativa nebyla, tak se ke sběru dat od různých uživatelů používají směrové vektory virtuální kamery ve scéně Unreal Engine projektu. Nakonec práce popisuje zpracování a vyhodnocení nasbíraných dat. Zásuvný modul a Unreal Engine projekt s experimentem i nasbíranými daty je dostupný na přiloženém médiu práce.

Klíčová slova virtuální realita, sledování pohybu očí, teplotní mapa, Unreal Engine, zásuvný modul, XTAL

Abstract

The purpose of this thesis is to explore the possibilities of eye tracking inside headsets for virtual reality, to analyse existing solutions, to create a prototype application that collects and visualises gaze data, and use it to create an experiment that collects the data from multiple users. The collected data are used to make resultant data that are analysed. A plug-in for *Unreal Engine 4* (UE4) is the form selected for the prototype application. To visualise the gaze data, heatmaps are generated for each object in the scene. This thesis describes the design of the *plug-in*, its implementation and its use to create an experiment of a traversal through a scene of the Czech National Bank Visitor Centre. The scene is in *Blender*, and the thesis describes the necessary modifications needed for a successful export to UE4 to achieve the correct functionality of the experiment. The result of this thesis is a working prototype of a *plug-in* for UE4, capable of collecting gaze data to produce heatmaps in real-time, that is not dependent on any eye tracking hardware. It also offers a description of the use of XTAL, which was not possible to make operational to collect gaze data due to the instability of its OpenXR runtime. There was no other alternative, so the forward vectors of a virtual camera in the Unreal Engine project scene were used to collect data from different users. Finally, the thesis describes the processing and evaluation of the data collected. The plug-in, the Unreal Engine project with the experiment and the data are available in the enclosed media DVD.

Keywords virtual reality, eye tracking, heatmap, Unreal Engine, plug-in, XTAL

Acronyms

VR	virtual reality
AR	augmented reality
XR	mixed reality
3D	three-dimensional
2D	two-dimensional
ET	eye tracking
UE	Unreal Engine
WS	World Space
IR	infrared
NIR	near infrared
UE4	Unreal Engine 4
UE5	Unreal Engine 5
HMD	head-mounted display
FOV	field of view
IPD	interpupillary distance
SDK	Software Development Kit
PoR	Point of Regard
RoI	Region of Interest
app	application
FPS	frames per second
CNN	Convolutional Neural Network
GPU	Graphics processing unit
SLAM	Simultaneous Localisation and Mapping
PCCR	Pupil Centre Cornea Reflexion
auto-PID	automatic interpupillary distance
Unreal	Unreal Engine

Chapter 1

Introduction

Virtual reality (VR) has experienced unprecedented growth in recent years. Since the release of *Oculus Rift DK1*, head-mounted displays (HMDs) for VR have registered annual advances in technical properties. The new devices operate at higher resolution and faster refresh rates, and also introduce new hardware features that broaden their set of tools. One of such is *eye tracker*; the subject of this thesis.

The desired effect is to use our eyes as a computer input, to interact in a virtual world, and to process our visual attention by collecting data about them. This hardware enables a VR headset to track the movement of its user's eyes to determine *gaze direction* in a virtual 3D space. The most current use has been found in measuring *interpupillary distance* (IPD) to adjust lens positions in HMD and in foveated rendering, which optimises graphics performance by lowering rendering resolution in the area around the point on display where the gaze, line of sight, intersects.

Performing an experiment inside a virtual 3D scene brings the advantage of knowing all the information about the objects contained in it, especially their location and rotation in world space. This allows gaze data to be collected and processed over time for each object separately using simple trace utilities (*raycasting*) of current game engines. The resultant data can determine whether the object had been seen or not, how many times it had been seen, and how long it had been gazed at. Such abilities can have numerous practical applications.

The prime example is the possible use in VR training for various professions. They all have different rules, work safety, mandatory steps during procedures, and best practises. All of this can be translated into test scenarios and then divided into multiple objectives that represent user actions. For illustrative purposes, imagine a situation where the driver of a car changes lanes on the road. The driver should look in the rearview mirror, activate a turn signal on the same side, look again, do the actual turn, and finally deactivate the signal. This sequence of actions can be programmed in a testing environment, and the action of whether the user gazed at the mirror could be registered with *eye tracking* (ET). These objective-based systems can provide constructive feedback because they can accurately identify whether a trainee followed the procedure or made a mistake. On a larger scale, a statistical model can be built for the entire experiment when sessions of different participants are analysed and compared.

Collecting information about subject's behaviour in a virtual environment using scenarios and objectives is an abstract concept that is not exclusive for VR training and could be applied to other use cases as well. For example, market research. Analyse what people look for in a supermarket and creating a heatmap around items that drew the most visual attention.

At the time of writing this thesis, there were very few software solutions that addressed the problem of eye tracking in virtual reality. Those that do exist are proprietary solutions with a steep price tag. This thesis will propose a possible solution of collecting and visualising gaze data that will be implemented with the use of current technology.

1.1 Goals

The aim of this thesis is to investigate the state-of-the-art of eye tracking technology, current devices, to explain the overall process of measuring eye movements and translating them into a 3D world, to explore the possibilities of using this technology to collect and visualise user gaze data in a predefined 3D scene, and to analyse the existence of other solutions that have already addressed this topic.

The main goal is to select an available eye tracking device and build a prototype application in a modern *game engine* that will be able to collect and visualise gaze data. The app will be used to create an experiment to test user behaviour within a virtual 3D scene in the same engine. Data from multiple participants will be collected and evaluated.

The Bachelor's thesis is divided into five core parts, where the first one will be devoted to the description of ET processes and principles not only used in the context of VR.

In the second part, the thesis will analyse other existing solutions and work done within the scope of this topic. Explore current hardware, software solutions, and analyse how gaze data can be collected and visualised.

The third part will propose a solution based on the previous findings and design a prototype application in a *game engine* of choice. The fourth one will focus on the implementation of the designed prototype, that will be used later by the last part, which will cover the creation of an experiment and the collection of gaze data.

Chapter 2

Eye-tracking technology

This chapter offers an overview of how the eye tracking process works and the methods used along with it.

The whole process can be divided into two parts: *eye detection* and *gaze estimation*. The first step interprets the captured images to detect the existence of an eye and to accurately find its position relative to the head. The next part, *gaze estimation* – uses the calculated eye positions to estimate *gaze direction*; orientation in space, also called *Point of Regard* (PoR). [1]

Both parts are explained in more detail in Sections 2.2 and 2.3, respectively.

2.1 Existing techniques

Duchowski recognises four categories of eye movement measurement methodologies: *electro-oculography*, scleral contact lenses, *photo/video-oculography* and video-based combined pupil and corneal reflexion. In his book, he briefly summarises and describes all of them. However, the last two categories will be covered together, because both represent video-based methods. [2]

All current VR devices with ET use PoR measurements. Techniques that are generally used most for that purpouse are video-based methods. This thesis will primarily focus on describing them, but briefly mentions the other ones first.

2.1.1 Electro-oculography

This was the most widely applied method to measure eye movements in the 1970s and is still used today in clinical settings. It is based on recording electric potential differences of multiple electrodes placed on the skin around an eye, see Figure 2.1a for an example. It offers good precision with high sampling rates [3].

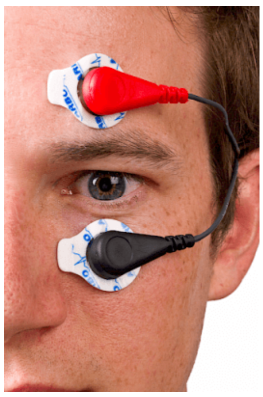
Eye movements are measured relatively to the head, but *gaze estimation* is possible if the head is fixed at one point in space or if a head position tracker is used in parallel. The devices used in *electro-oculography* belong to a family of intrusive devices; physical contact with the user is required [3]. Its intrusive nature allows eye movements to be recorded even if the eyes are closed.

2.1.2 Scleral contact lens

Techniques based on scleral contact lenses are the most intrusive of all, but fundamentally, this makes them among the most precise[1] in terms of eye movement measurement.

The method uses special contact lens with a reference object attached to it. This entire gadget is worn directly on the eye. During the nineteenth century, the first attempts used lens alternatives. One of such was a plaster ring, which was attached directly to the cornea. The contact lens must be larger to cover both the cornea and the sclera to prevent slippage, due to the increase in weight added by the extra object. Among the mechanical and optical attachments used, the most popular were reflecting phosphors, line diagrams, and wire coils in magneto-optical configurations. In particular, a wire coil is used in the principal method to measure eye movements using an electromagnetic field. An example of a scleral lens with wire coil can be seen in Figure 2.1b.

The biggest downside of a scleral contact lens is the discomfort of wearing one. These methods also provide eye positions relative to the head.


(a) An electro-oculography device. [4]

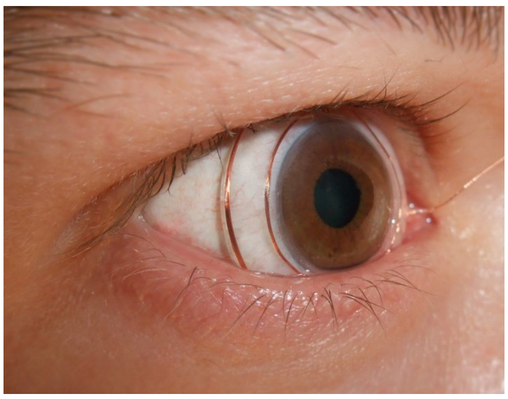

(b) A scleral contact lens with wire coil. [5]

Figure 2.1 Examples of intrusive eye tracking devices.

2.1.3 Video-based methods

Photo and *video-oculography* group together methods that record eye images with the use of cameras or video cameras to measure eye movements. The recording device is either attached to the users head; head-mounted; or placed near the user; table-mounted. Devices that are used in the these methods are non-intrusive, even if the device is mounted on the head, because the actual measurement is not made by direct physical contact with the user.

Hansen has described video-based methods and their general process in great detail. The publication is, despite its age, still relevant today because the methods presented have not changed. [1]

The idea behind video-based eye movement measurements is to inspect distinguishable eye features from captured eye images, which must differ enough to clearly distinguish between eye and head movements. These are the appearance and shape of the pupil, iris, and cornea.

The techniques that use these features are implemented using computer vision. It is possible to process them manually frame-by-frame. Doing so becomes more time-consuming, inefficient, and unreliable with each added frame. Multiple errors can be made during the process.

The most widely applied method for measuring eye movements is the one based on corneal reflexion, example shown in Figure 2.2, which involves the use of – at least – one camera and

[1]The accuracy is measured within arc-second units

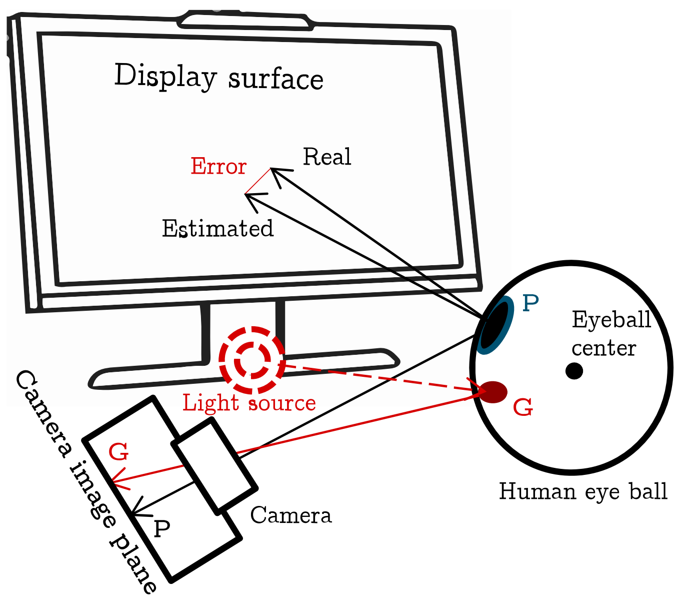


Figure 2.2 Typical video-oculography setup using method of corneal reflection. [6, p. 16497].

a near infrared (NIR) light source to produce glints on the eye cornea. The relative position of the glint and the pupil centre is measured to estimate *gaze direction*. Infrared (IR) light is used because it is invisible to the naked eye and cameras can capture it. This property is useful for controlling the lightning conditions in the environment. It is more suited for indoor setups, where natural light can be controlled and IR rays can bounce around. The natural light can be also utilised for the same purpose, but it offers a poorer contrast [1].

It is impossible to make an absolutely precise gaze estimate; always a certain deviation from the real line of sight must be taken into account, as shown in Figure 2.2. P represents the pupil on the human eye ball and G is the glint created by the light source. The camera image plane is the captured eye image, in which P and G indicate the position of the pupil and the glint. [6]

ET process

Each *video-oculography* system must go through the same process, shown in Figure 2.3, to estimate *gaze direction*. In the initial step, this procedure requires image data. At least one camera is needed. The image is processed to determine the existence of the eye and to detect its position, which can be used directly by an application or further used to estimate *gaze direction* – PoR, which is the final output an application can receive from the system. For this to happen, the position and rotation of the head needs to be tracked.

Tracking head position

One possible solution is to simply sacrifice a degree of freedom and fix the head in one place in space – resting a chin on a raised support. It is functional but not optimal, because there are techniques that solve this. Another device or system dedicated to tracking the head position needs to be used in parallel. If the system is table-mounted, the head position is already handled by some *gaze estimation* algorithms. Meanwhile, this is impossible for head-mounted systems, because they move simultaneously with the head, so the *gaze direction* behaves the same. Another device or system dedicated to tracking the head position needs to be used in parallel.

There are several positional tracking systems. Some track the absolute position of the user in a room, such as VICON tracker, Lighthouse from Valve, ART trackers, OptiTrack, . . .

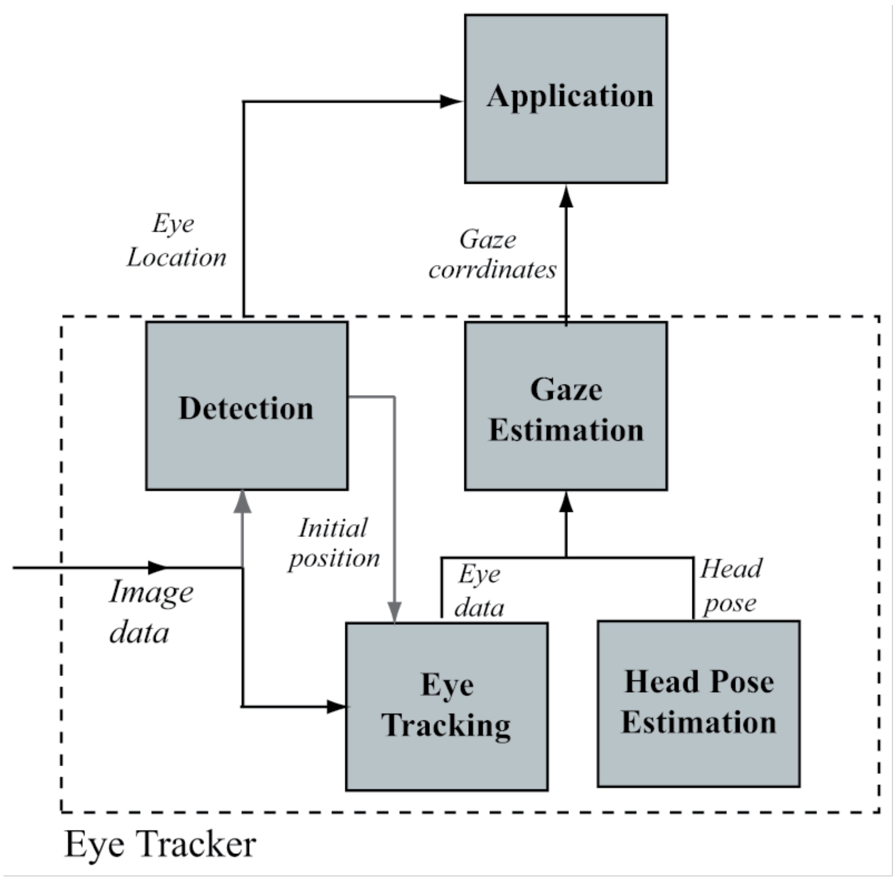


Figure 2.3 Components of eye tracking process. [7]

Other systems can detect the position of the head while being head-mounted. This is a simultaneous localisation and mapping (SLAM) problem, which is solved, for example, by the Intel T265 tracking camera. However, this is beyond the scope of this thesis.

Table-mounted devices estimate PoR on a surface of interest – usually a monitor or a TV screen – which might be too restrictive because they cannot move with the user and may experience difficulties registering eye positions from angles invisible to the camera. Wearable eye gaze tracking devices use virtual scene cameras to measure gaze in 3D space. [3]

2.2 Eye detection

This section derives information mainly from Hansen [1]. The purpose of *eye detection* is to determine the existence of the eye and its position from the captured eye images. It is crucial, in order to successfully perform it, to find an eye model that is flexible enough to account for changes in the eye's appearance without being computationally too expensive. Techniques use computer vision algorithms to analyse captured eye images to obtain information that describe the eye's status, appearance, lighting conditions, and reflectivity. Specifically, the viewing angle, head position, occlusion by the eyelid, degree of eye openness, and position and colour of the iris. These are very specific features that drastically change the appearance of the eye. Eye models are determined by approaches that use the shape or appearance of the eye.

2.2.1 Shape

These approaches create eye models using shapes of eye features. Their important characteristic is the ability to handle changes in the shape, scale, and rotation of the eye. Models are defined by a geometric shape that is either rigid or deformable, based on its parameters.

Rigid shape

Simple elliptical shape is an computationally efficient rigid model that takes advantage of the eye features that look elliptical; iris and pupil. The current state of those features under different viewing angles is found by edge detection algorithms. The iris is detected by finding the limbus, which is the boundary between the iris and the sclera, and the pupil by detecting its edge with the iris. The variability of these features is restricted to two degrees of freedom that describe eye movements; pan and tilt. Higher-contrast images are preferred for easier and more precise edge detection.

Complex shape

More detailed eye models can be made for greater precision. Methods based on a deformable eye model use for the eye description generic deformable template, which is modifiable with parameters. One method is used to represent the eyelids and the iris. The former is described by two parabolas with 11 parameters, while the iris is a simple circle. Another method describes the deformable eye with two semiellipses that are defined by six parameters.

Methods based on a deformable eye model offer an accurate and generic representation of the eye, but they are computationally very demanding and require high contrast images that are taken from a close distance to the eye for their proper operation. Algorithms solving these issues incorporate energy minimisation processes of computer vision. The same approaches are applicable to face recognition.

2.2.2 Appearance

Appearance-based approaches detect and track eyes directly from captured images, as opposed to shape approaches. The eye and its features are described by the colour distribution in the image or by the results of *image filters*. The computer vision techniques employed do not require any knowledge about the eye itself. These are actually able to model absolutely anything. The requirement to train a model is to use a sufficiently large set of images that represent the appearance of the desired object.

Methods based on these approaches can have problems with rotation, scale, and detection of the eye. The reason for this may be the insufficient amount and diversity of training data that do not reflect different types of eyes, head orientations, or lighting conditions.

Information about the appearance of the eye is extracted from captured images either directly or by their transformation, which can enhance the features of the eye or suppress differences in illumination.

Image filtering

Image filtering are techniques used to modify raster images, either over the entire image or just in a selected area. Filtering can be done in two ways, either by filtering only one pixel value or by adjusting a pixel, based on its near surrounding neighbours. These techniques include, for example, brightness and contrast adjustments, gamma correction, thresholding, blurring, edge detection, and convolution. [8]

Eye features

Captured eye data can be transformed by filters. Filter responses to a given image can also be used for individual eye features. Some are more appropriate than others. Limbus boundary detection is easily implementable but offers low precision [3]. Generally, explicit eye edge detection may not always be correctly detected due to changes in light, image focus, occlusion, or eyelid cover.

Eye images taken from a reasonably close distance allow the pupil to be a reliable feature. It depends on the pupil, and the iris appears darker than in other areas. This might require higher-contrast images. One method uses an iterative threshold algorithm that searches for two dark regions. However, this method alone is insufficient in some cases. It could confuse the pupil with areas with dark shadows, eyebrows, or even areas of individuals with darker skin. There is an improvement for this method that uses an NIR light source.

2.2.3 Dark and Bright pupil

These are two methods that extend the eye detection method by localisation of the pupil with an IR light ray. It is used to increase the pupil contrast in captured images for more robust eye detection.

The methods depend on the angle of light entering the pupil. When the IR light bounces back from the eye to the camera, the pupil brightens up. On the contrary, when the light is at an angle away from the camera, it creates an effect where the pupil appears much darker.

This Czech thesis presents an implementation of the dark pupil method using a *convolutional neural network* (CNN) [8].

2.2.4 Purkinje images

Another characteristic of the eye that is observed is the behaviour of how light reflects off its surface. These features were found by Jan Evangelista Purkyně. They are named after him and called *Purkinje images*. In the context of ET, IR light is used to produce different types of reflexions on the eye, which are then detected by algorithms.

Exactly four distinct *Purkinje images* can be recognised. First one (P1) is reffered to as a glint, which is reflexion of the outer side of the cornea. Second (P2) is the reflexion from the back side of the cornea. Third (P3) bounces off the front side of the eye lens. Fourth (P4) image captures the reflectivity of the back side of the eye lens. [9]

2.3 Gaze estimation

Determining gaze is to be understood as the gaze direction or PoR, which is a 2D coordinate on a surface. *Pupil Centre Cornea Reflexion* (PCCR) is a method that uses an NIR light source to produce the first Purkinje images while simultaneously darkening the pupil using the dark pupil method. This results in a very contrasting image of a glint and a pupil. These features are easy to detect, so their positions are determined. It is the most widely used ET method to date [10]. Gaze is estimated by their relative position, which is used by two different methods. [3]

However, the position and orientation of the head need to be taken into account for systems that do not operate with a fixed head. In these cases, the head movement must be compensated for. This problem is not solved with HMD systems because they move simultaneously with the head. [6]

2.3.1 2D regression

2D regression is a group of methods that use the position of the pupil and the glint of the eye to map directly to a PoR. This mapping is expressed by some polynomial function or by a neural network. There are 2D regression methods that are capable of head compensation during polynomial mapping [6]. Some are able to address this by adding an additional camera, which requires stereo calibration [1].

For these methods, it is essential to use a calibration for the polynomial function. It is often based on several points on the screen that are spaced around. The points cause the eye to look in a certain direction. The camera records the eye. From this eye image, the position of the pupil, the glint and their relative vector are determined and assigned to a certain gaze vector or PoR on the screen according to a specific calibration point. This operation finds the correct parameters for the polynomial function from them. The accuracy of gaze estimation is improved with a larger number of calibration points. [6]

2.3.2 3D model

Then there are methods that create a geometric 3D model of the eye from the detected features, and the estimated gaze originates from the centre of the cornea as the optical axis of this 3D model. These methods utilise the similarity of the shape and size of the eyes. [6]

For the complete construction of the model, knowledge of the camera position and light is required, which can be obtained by calibration. For cases where the head position keeps changing, the head position must be constantly evaluated. External positioning systems can be useful for this. The basic device that uses a 3D model to estimate gaze is one IR camera and one IR LED. Such a solution might require calibration. However, there are many other solutions that add multiple LEDs and cameras to the system that might no longer require a calibration. If it does, only once in a while. For example, if multiple users are using the device, each would need their own calibration. As the number of cameras and LEDs increases, so does the accuracy and quality of the measurement, while the need for calibration decreases. [8]

Chapter 3

Analysis

This chapter covers the analysis of current gaze tracking practises, systems implementing ET, the latest VR devices with ET and the capabilities of Unreal Engine with possibilities of implementing ET in it.

3.1 Gaze tracking

ET technology combined with VR allows one to estimate the gaze direction in a 3D world, which is a 3D vector. Various techniques that solve the manipulation, classification, and storage of this line of sight are part of gaze tracking.

3.1.1 Fundamentals

The way an eye moves can be described as a continuous sequence of two alternating actions; fixation and saccade. [1, 6]

Fixation is a stable but non-static gaze state that rests on a small area for a period of time. A gaze can only be classified as a fixation if it remains in the area for a defined minimum threshold duration – usually 80 ms.

Saccade is a gaze state that indicates rapid movements of the eye between fixations. It can be classified as such if the gaze leaves the small area prior to fixation classification.

Fixation is the main metric used in ET, but it requires context. In order to use gaze to determine fixation within a 3D scene, it is first necessary to segment the scene into several logical sections.

Region of Interest

This segment is called *Region of Interest* (RoI), which provides the context for the fixation, i.e., the gaze target. These can be defined as large areas with multiple objects or, alternatively, as small details. Each virtual scene contains information about its objects and their absolute positions in its coordinates. It is reasonable to create RoIs according to individual objects in a given scene. [9]

An ET experiment consists of subtasks during which fixations and saccades at different RoIs are captured over time to determine when and where the subject gazed at. This information can be used in the creation of further structures that provide more context to the subject's behaviour during an experiment. [11]

Scanpath

One of these is *scanpath* [6], which consists of an alternating sequence of fixations and saccades. These are not only useful for storing the sequence of eye movements over time, but also can be used to infer the duration and distance between two different fixations.

With its combination, more precise information is obtained about each *region of interest*, from which one can infer how much time elapsed before the first fixation in the RoI or how many and which regions were visited before. Another related metric is *gaze duration*, which sums the duration of all fixations in a single RoI.

3.1.2 Gaze data

The data are constructed in real time during the experiment or after its end. Everything depends on the deployed hardware. It is recommended to have all available performance ready to use for the experiment itself. For later processing of ET data, it is crucial to record the gaze data first. Sources used for this section [9, 11]

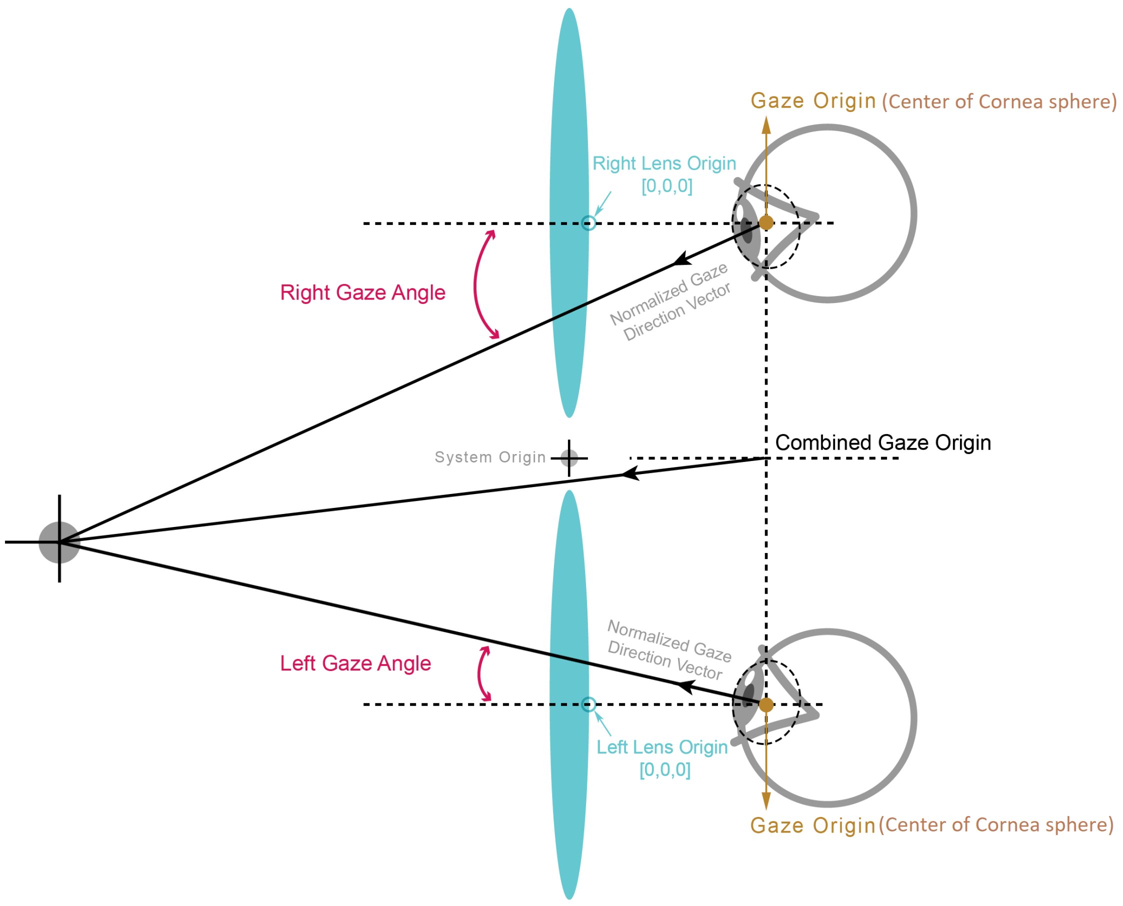

Gaze Origin (Center of Cornea sphere)
Right Lens Origin [0,0,0]
Right Gaze Angle
Normalized Gaze Direction Vector
Combined Gaze Origin
System Origin
Normalized Gaze Direction Vector
Left Gaze Angle
Left Lens Origin [0,0,0]
Gaze Origin (Center of Cornea sphere)

Figure 3.1 Detailed scheme of gaze data [12]

Gaze structure

Gaze data are the result or output of an ET device, but not every time the data are suitable for the use in a virtual scene. Some eye trackers produce 2D coordinates on a display surface that represent PoR [13]. If the produced data are to be used in a virtual scene, it should be a collection of 3D vectors. This is illustrated in Figure 3.1. The gaze itself from a single eye is represented by two 3D vectors; normalised *gaze direction* vector and *gaze origin* coordinates.

Both are in a coordinate system that is related to that particular eye. In this specific situation, the origin of the coordinate system is a fixed point that is not related to the eye itself because it can account for variations in the eye positions of different people. Figure 3.1 shows the use of a lens in HMD for VR as the origin of the eye's coordinate system.

Point of Fixation

Subjects usually have two functioning eyes, so stereo gaze data can be used in their situation. Two separate coordinate systems, eye positions, and gaze directions. In the best case, the intersection of these two vectors in world coordinates can be used to identify *Point of Fixation* (PoF).

However, the problem is that the absolute precision of these vectors is not guaranteed due to limitations in eye detection using video-based ET devices, which return acceptable results only for perfect calibrations. The farther in a scene the eye gazes, the more precision is required. In this case, the accuracy must be below 0.5 degrees.

The difference between *fixation* and PoF is that fixation describes the type of gaze, and PoF is the point in space where fixation occurs.

Combined gaze

An alternative to stereo gaze data in multiple implementations of ET in HMDs is a combined gaze vector. The combined gaze origin is the average of the two origin coordinates, and the direction of gaze is the normalised sum of the two gaze vectors. Each vector calculation is performed in the ET device coordinate system.

Data processing

To search for fixations using the combined gaze in the scene, one has to resort to collisions. Sending a ray in world coordinates from an eye into the scene – *raycasting*. PoF is the first object (RoI) hit by the ray.

The search for fixations can be done in real time during the experiment or sometime after its end. In the case of offline evaluation, it is first necessary to gather the scene information correctly together with the gaze data during the experiment. The essential data consist of time in seconds or milliseconds, a 3D coordinate that reflects the position of the scene camera in the world, its rotation, and the gaze origin with its direction; two 3D vectors in world coordinates.

These are saved in a CSV file or in an internal database. During prototype testing, it is more convenient to save data in CSV because the file can be deleted at any time after a failure, as opposed to deleting non-valid entries from a database. Performing offline computations is meaningful only when the gaze data are processed and visualised in several different ways that clearly consume computational power. If a search for PoF is performed alone, there is no reason not to compute it in real time.

3.1.3 Visualisations

Analysing the existing solutions later in Section 3.3 found that there are generally two ways to visualise gaze data in a virtual environment. Many analyses of gaze data consist of recording them in external files and processing or visualising them with a help of a statistical programme. The basis for all visualisations are PoFs in 3D space. These points are then transformed or classified into more specific structures.

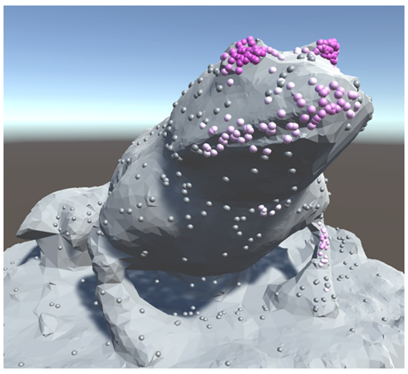
Heatmap is a graphical representation of data where each value on a map is represented by a colour from a predetermined spectrum. Typically, this is a transition from blue, representing the minimum value, through green and yellow to red, which indicates the maximum value. In the context of fixations, higher heatmap values are indicative of a denser concentration of PoFs or their individual strength; the duration of fixation.

Gaze trail is a visual representation of *scanpath*, where PoFs are displayed by a simple cube or sphere primitive. Saccades connect these primitives in space with lines.

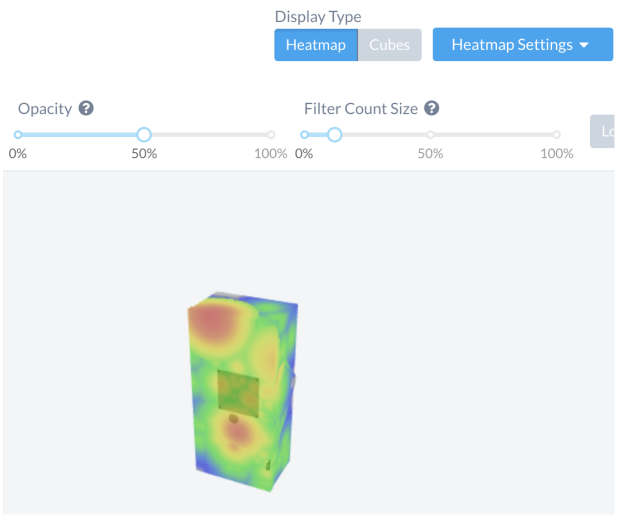
This interesting study from 2005 explores attention search in a virtual environment [14]. This is an experiment that explores visual search for different objects in a virtual three-room apartment with the participation of several subjects over multiple days. They used a custom HMD with lower resolution to collect the data, given the age of the study. Eye images, camera view, and participant's position in the scene were recorded for an evaluation. The whole situation was reconstructed offline for analysis.

The PoF was obtained using an algorithm that took the eye position from an eye image and used its screen coordinates to extract a 60x60 pixel square from the camera scene footage. From these few pixels, it was evaluated which object occupied the most pixels. The same object's location was used as the 3D fixation coordinate.

It is possible to visualise 3D data by condensing them onto a 2D map. They created several graphs and 2D heatmaps from the gathered PoFs. For example, the floor plan of an experiment scene can be overlaid with a heatmap that describes the spatial intensity of gaze data.



(a) 3D Heatmap points [9, p. 52]



(b) Object texture heatmap [15]

Figure 3.2 Example of two different heatmap approaches.

This paper [13], which is closely related to a thesis by the same author [9], visualises heatmaps using 3D points in space that are coloured differently according to the intensity of a given fixation, can be seen in Figure 3.2a.

Cognitive3D software visualises gaze data using heatmaps and visual trail. Heatmaps are generated in a different way. They draw it on the wireframe of a 3D model – Figure 3.2b – assumably onto a texture. The algorithm they use to do this is proprietary. For real-time visualisation of gaze data, without the need to collect and evaluate the data offline, the method of drawing heatmaps on objects is more suitable.

3.2 Current hardware

ET devices implemented in VR HMDs are portable devices that have only recently seen their first prototypes. The same applies to the first truly usable VR headsets that have started to appear since Oculus' first attempt with its *Rift DK-1* headset. This was immediately followed by the *DK-2* version. HTC joined in on this with their VIVE headset. They were all released in a short period of time before 2016, which is exactly the year when all the aforementioned were extended with an SMI eye tracker. [13]

Unfortunately, SMI announced that it discontinued the production and support of its ET devices when it was acquired by Apple in 2017 [16]. Regardless of this whole situation, Tobii created an ET Development Kit for HTC VIVE that was not bundled with the heatset itself. It had to be purchased separately and mounted inside a very confined space around the lenses [17]. Modern headsets are made with ET integrated into them by design. Manufacturers have two options. They either develop ET devices themselves as their proprietary solution or leave it to a third-party. One of them is Tobii, which dominates the ET market. [9]

3.2.1 Tobii

This company is exclusively specialised in eye tracking. Since their inception, when they released the world's first plug-and-play ET device in 2002 [18], they have grown to become a global leader in the industry.

Design

Current Tobii eye tracking systems are based on non-intrusive video-based tracking that uses the PCCR method. Their implementation uses multiple NIR light sources to create reflexion patterns, several glints, on the cornea and pupil, which are recorded by cameras. They use an advanced image processing algorithm based on a physiological 3D eye model to detect the eye and estimate the gaze. [10]

Most of their devices use the dark and bright pupil method for eye detection, except for wearable devices that use only the dark pupil method. All of their devices use the method mentioned using a 3D eye model for gaze estimation. [19]

Wearable device

An example of their wearable eye trackers is Tobii Glasses 3, shown in Figure 3.3. The device is designed to handle real-world ET research. A robust system is in place, consisting of eight NIR illuminators for each eye separately that create glints. The single eye with glints is then captured by two very small cameras with a sampling rate of 100 Hz. A one-point calibration is performed before measurement. The device is also equipped with a full HD camera, which is located in the beam of glasses. It is a portable device that does not require a connection to a PC, as the product includes a recording unit that captures video from the camera and simultaneously collects ET data to an SD card. The accuracy of the measured data is around 0.6°. The data are

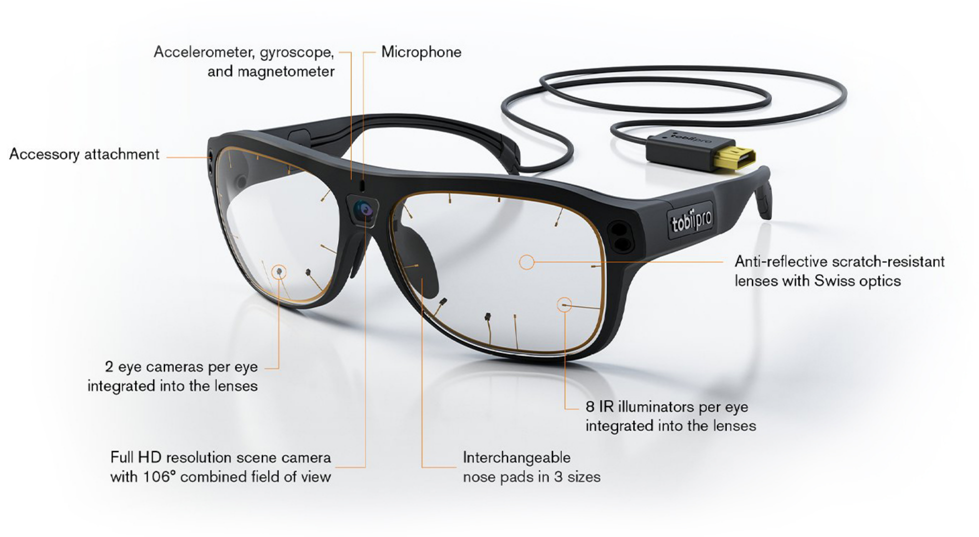

Accelerometer, gyroscope, and magnetometer
Microphone
Accessory attachment
Anti-reflective scratch-resistant lenses with Swiss optics
2 eye cameras per eye integrated into the lenses
8 IR illuminators per eye integrated into the lenses
Full HD resolution scene camera with 106° combined field of view
Interchangeable nose pads in 3 sizes

Figure 3.3 Description of Tobii Glasses 3. [21]

processed and visualised using their analytics software, Tobii Pro Lab. Their Tobii Pro Glasses 3 API allows for a custom implementation. [20]

Table-mounted device

Tobii also develops table-mounted devices, one of which is the Tobii Pro Fusion. It is an ET device that consists of two cameras with a sampling rate of 250 Hz. The dual camera setup is used for better head movement tolerance. The accuracy of this device is 0.3°. Gaze data can be collected and used in their proprietary Tobii Pro Lab software or with the Tobii Pro SDK, that allows for integration in C, Python programming language,.NET framework or *Unity* game engine. [22, 23]

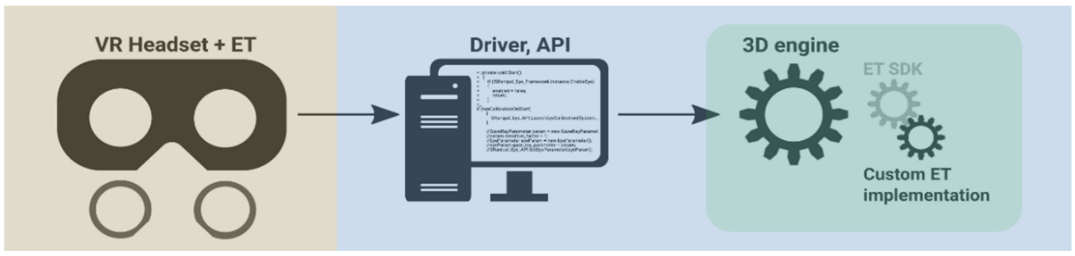


Figure 3.4 Interconnection of ET hardware with SDK. [13, p. 4]

3.2.2 Integrations in VR HMDs

In addition to Tobii's own eye trackers, the company has partnered with various VR headset manufacturers to provide integration of their ET units. These are HTC, PICO, and HP. Specifically, the headsets with integrated ET from Tobii include HTC VIVE Pro Eye, HP Reverb G2 Omnicept Edition, PICO Neo 2 Eye and Neo 3 Pro Eye. A comparison of these devices can be found in Table 3.1, which shows their specifications. Frequency of the output gaze data, measurement accuracy, calibration method, trackable *field of view* (FOV), available *Software Development Kit* (SDK), type of output gaze data, the state of OpenXR standard implementation, and SDK support for the two major game engines.

	HTC VIVE Pro Eye [24]	HP Reverb G2 Omnicept Edition [25]	PICO Neo 3 Pro Eye [26]
Frequency	120 Hz	120 Hz	60/90 Hz
Accuracy	0.5°–1.1°	< 1.0°	< 1.0°
Calibration	5-point	9-point	5-point
Trackable FOV	110°	114°	101°
SDK	HTC SRanipal	HP Omnicept SDK	Pico SDK
Stereo gaze	Yes	Yes	No
Combined gaze	Yes	Yes	Yes
OpenXR	Yes	Yes	Ready
Unreal, Unity	Both	Both	Both

Table 3.1 Specification of the latest VR headsets with Tobii ET integration.

Software Development Kit

HMDs communicate with software using a driver. SDKs are used to add the communication functionality to the software. This set of development tools is provided by the hardware manufacturer. In the case of VR implementations, the SDK is in the form of an *Application Programming Interface* (API) that returns the requested data to the application from some VR runtime that is part of the hardware driver. As Figure 3.4 illustrates.

Tobii SDK

One SDK is available for all HMDs with a Tobii eye tracker; Tobii XR SDK. The strength of using this is the unified access to the data across all devices. On the other hand, it offers a stable release only for the *Unity* game engine. *Unreal Engine* SDK is still in beta version[1] and support for native C development is only in Alpha. In contrast to all the dedicated device SDKs listed in Table 2.1, which offer full support for the mentioned game engines and native C. [27]

[1]does not work with HTC VIVE Pro Eye

3.2.3 OpenXR standard

Until recently, for an application to be able to use a specific VR device, a proprietary API was required to communicate with its hardware driver. Support for different devices in one app required custom driver integration. However, the Khronos Group has proposed a new standard to fight this fragmentation; *OpenXR*, which provides high performance cross-platform access to *Augmented Reality* (AR), *Mixed Reality* (XR) and *Virtual Reality* devices and platforms.

This includes not only HMDs, but also VR controllers, hand tracking, and eye tracking. New applications are now able to share the same interface to communicate with several different VR, XR, or AR devices, as the schema on Figure 3.5 shows. OpenXR is not a library or an implemented runtime, but a specification of how that runtime should look. The implementation is done by XR device manufacturers themselves. [28, 29]

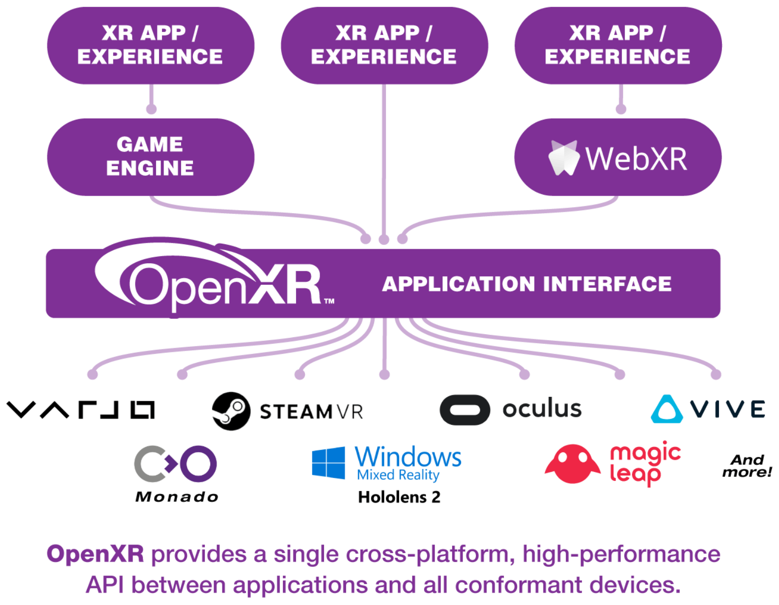


Figure 3.5 Schema of OpenXR cross-platform access to different XR devices. [30]

An important aspect of OpenXR is that it uses a Cartesian right-handed coordinate system to describe the space in which all vectors are represented; the x coordinate points to the right, the y points up, and the z coordinate points backward ($-z$ points forward into the scene). [31]

3.2.4 Varjo

Varjo is a Finnish manufacturer of VR and XR HMDs. Their latest VR devices are VR-3 (Figure 3.6) and the Aero headset. All Varjo devices are equipped with Varjo's own proprietary ET solution, not based on Tobii. They claim that their method is "the most accurate eye tracking ever integrated into a VR device". [32]

Method

It is a method based on 2D regression that is enhanced by the type of illuminators and camera quality. It uses a special custom IR illuminator shape that produces complex-shaped glints, instead of dots.

The robustness of this solution lies in the ability to use computer vision algorithms to distinguish one illuminator from another by examining the orientation of its glint. Combined with

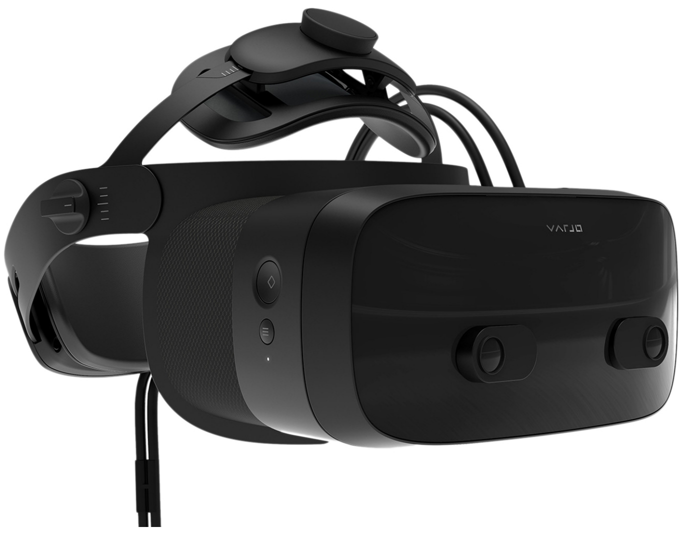


Figure 3.6 Varjo VR-3 virtual reality headset. [37]

IR cameras that capture eye images at 100 *frames per second* (fps) at a resolution of 1280x800 pixels, the accuracy of eye measurement is guaranteed to be sub-zero degrees even when the eyes are looking at extreme angles or when glasses are present.

Capabilities

Both headsets are capable of producing combined and stereo gaze output data at 200 Hz. They include an automatic PID (auto-PID) capability to adjust the lens distance and foveated rendering after a single-point calibration. What VR-3 has over the newer and lighter Aero is a built-in Ultraleap camera for hand tracking, which is visible in Figure 3.6. Headsets use the native Varjo SDK or OpenXR runtime [33] to communicate with apps. Offers support for implementation in native C and integration into Unreal Engine and Unity game engine. [34, 35]

Gaze data output is natively in the left-handed coordinate system. The vectors are then represented in a space where x points to the right, y points up and z points forward in the scene; compared to OpenXR, the z coordinate is flipped. To start ET, the device must first be calibrated. This can be done in three different ways. Either a five-point calibration, which uses statistical data collected from previous calibrations by other users to enhance the calibration result, or a ten-point legacy calibration, which does not include such a feature. One point cannot be used to initiate ET, it only serves to activate foveated rendering. [36]

Varjo driver has built-in tools to record gaze data. This feature records them simultaneously with a video of what the user has seen. The output is a .csv file that can be used to visualise the data collected in the recorded video. The CSV contains generic gaze data along with their transformation into video coordinates. The generic data contain the origin and direction of the gazes of the left and right eye and their combined gaze with a raw timestamp of when the data were recorded. The second data in relation to the video are transformed into 2D coordinates (PoR) and stored together with the timestamp relative to the start of the video. [38]

Immersive Virtual Reality

Another Varjo device is the XR-3 headset for mixed reality, which is almost identical to the VR-3 but includes the feature of a video feed of the external world. This headset uses two cameras to record video that can be modified and rendered back on the internal displays. [39]

Immersive Virtual Reality (IVR) is a 3D projection used in Religious Studies testing Laboratory build by Olomouc Palacký University. [40]

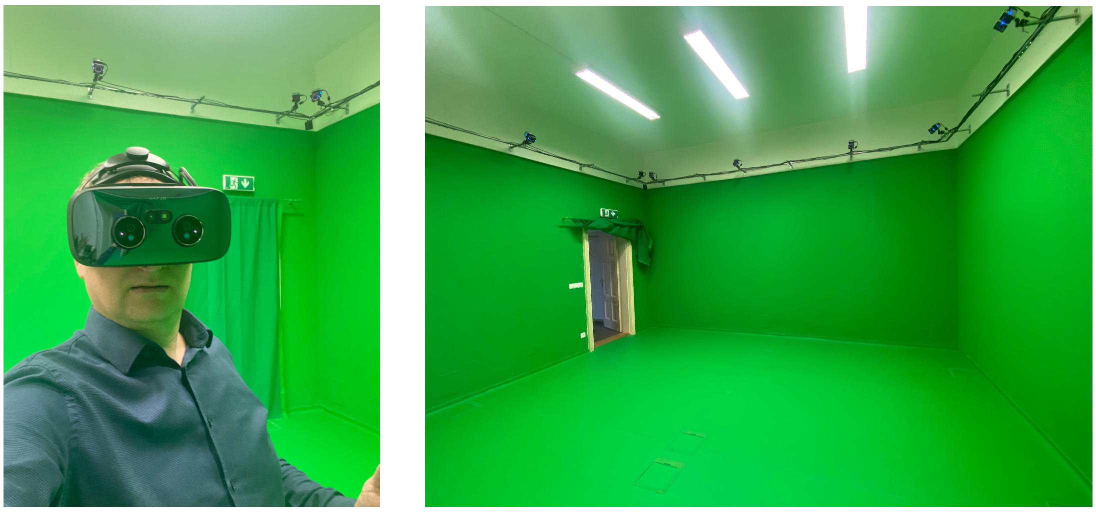

(a) Varjo XR-3 headset in use. **(b)** Interiour of the laboratory.

Figure 3.7 Mixed reality laboratory for religious studies in Olomouc Palacký University.

This Czech lab uses the aforementioned XR headset and its ET capabilities to record and study human behaviour during religious processes. The headset records a video of the lab, and then the green screen is keyed out during the post-process. The result is used to overlay the virtual scene that is displayed later inside the headset. They use VICON's tracking system to track the position of the headset. The system uses a multiple camera setup, some of which can be seen in Figure 3.7b.

3.2.5 Vrgineers XTAL

Vrgineers is a Czech-American company that has been making VR headsets since 2016. Their hardware is called XTAL, and all of its versions include built-in ET hardware. The latest XTAL is in version 3 [41], Figure 3.8, which also exists in a version for XR that includes additional cameras that project the outside world inside.

Features

Headsets consists of several features. They offer auto-IPD, hand tracking with the help of Ultraleap cameras, and an ET system that consists of two arrays of several IR light dots, one for each eye, to which a single IR camera is paired. An IR camera has a resolution of 1280x800 pixels, which is captured at 110 Hz frequency. ET operates natively at 120 Hz, but it is possible to increase the frequency of the eye tracker's output data up to 210 Hz. [42]

Runtime

At the time of writing, the XTAL headsets communicated only using their stable native VRG runtime, which uses this ET system only for IPD measurement and foveated rendering. This was not entirely clear at the beginning of the investigation and was believed to be capable of returning gaze data because the source code of their SDK included gaze data structures similar to those presented in Section 3.1.2. However, Vrgineers support later informed that they were developing another runtime that was capable of this.

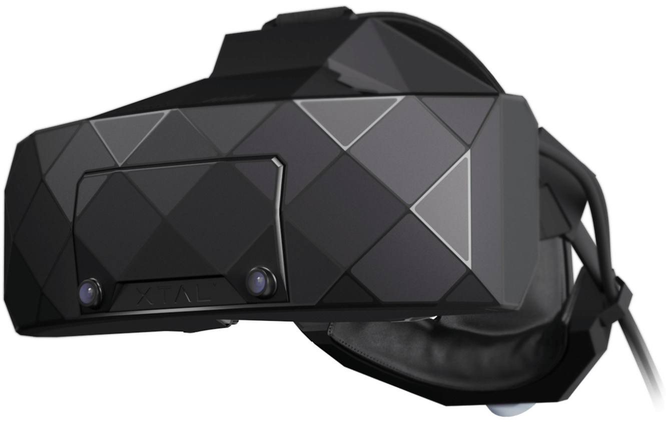


Figure 3.8 VRGineers XTAL 3 VR headset. [44]

Their goal is to convert the entire native runtime to OpenXR, which is currently in beta and already offers left, right, and combined gaze data according to the OpenXR specification. The native runtime offer support for native C development and integration to Unreal and Unity. They tested the beta OpenXR runtime on Unreal Engine versions 4.27 and 5. [43]

Eye tracker specification

In this Bachelor's thesis written under the Faculty of Electrical Engineering, CTU in 2020, a prototype of eye detection and gaze estimation using the ET system of the XTAL headset was implemented. The author used the dark pupil method to detect the eyes by scanning the pupil in the recorded eye images. Gaze estimation was performed using a 2D regression method. [8]

At this point, it is not clear whether the ET implementation from the Bachelor's thesis is present in the current driver or if it has undergone significant changes. It is unrealistic to assume the accuracy of the measuring algorithm because the manufacturer has not yet provided it.

ET does not work in XTAL without calibration. While a VR application is running, one of the two calibration options in the VRG runtime must be triggered. The faster one is the one point calibration, which is used to measure the PID and enable foveated rendering, but can also be used to enable ET itself. This option uses for calibration a virtual mathematical model of the user's face. For more accurate results of the measured gaze data, the more advanced calibration, consisting of seven points, should be used. [45, 43]

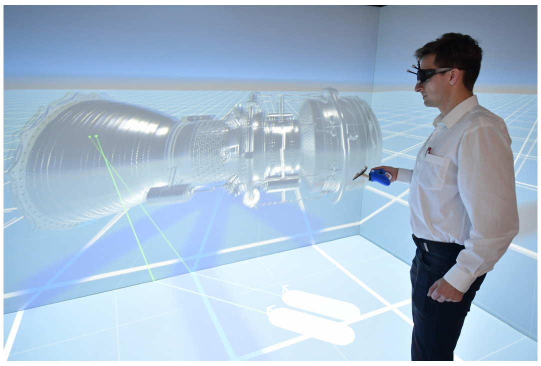
XTAL VR Training

Vrgineers has partnered up with the U.S. Air Force to build custom simulators for training fighter pilots. The simulator can be seen in Figure 3.10. When the pilots sit inside, they have the XTAL 3 XR headset on. This is an example of VR training that mimics the environment of various fighters such as F-15, F-18, and F-35 as closely as possible to train muscle memory and situational awareness of pilots. [46]

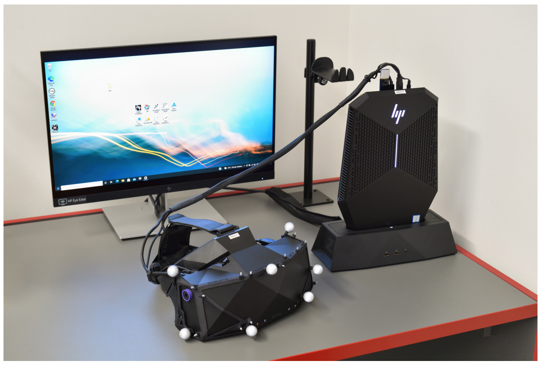
3D CAVE

The room with CAVE 3D is a laboratory of the Faculty of Mechanical Engineering of CTU. It consists of a multiprojection system called CAVE, which is used to visualise scenes and scientific data. The CAVE simultaneously projects stereoscopic images onto four cube faces; the front, right, and left walls with the floor, Figure 3.9a. More specifically, it is used to display and verify virtual prototypes, where real physical parts of the structures are compared to the virtual prototype. [47]

The very second part of this lab is a corner with four XTAL headsets that run the same application as the projection system. The headset can be walked around the lab by attaching it to an HP backpack. Figure 3.9b.



(a) CAVE 3D multi-projection system.



(b) XTAL with HP backpack.

Figure 3.9 3D CAVE laboratory at CTU FME. [47]

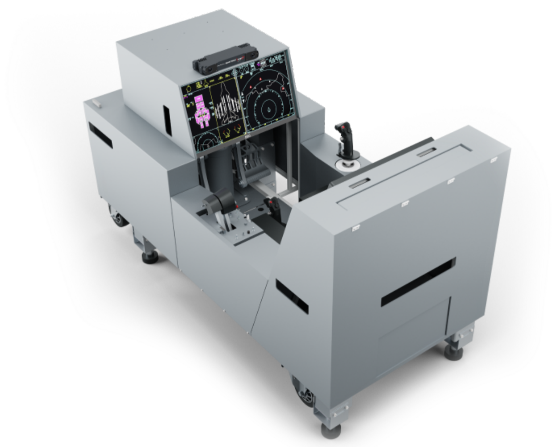


Figure 3.10 VRGineers fighter jet cockpit simulator. [48]

3.2.6 Future possibility

Another headset currently in development is the successor to the Playstation VR headset, which is designed for the PS5 console; the PSVR2. A huge advantage of the original version was that it was one of the most affordable VR headsets on the market. It admittedly did not have the best specifications, but it did not cost more than $400 when it was released. Compared to the other headsets that could be purchased for at least double that amount. Although the headset was made to be used exclusively on a Playstation platform, it is possible to communicate with it on a PC via unofficial drivers.

The new PSVR2 has dramatically improved technical specifications over its predecessor. The internal OLED panel offers 2000x2040 pixels for each eye with a refresh rate of up to 120 Hz, and an FOV of 110 degrees. Most interestingly, it includes an eye tracker. Sony's implementation will likely be proprietary to keep the cost as low as possible. The setup is conventional with IR illuminators and one IR camera for each eye. The main reason they want to implement ET is to add additional input for new games and for the foveated rendering capability, which is mainly to free up PS5 rendering performance. It should offer developers of new PSVR games access to the same gaze data structure as the previous ET headsets. [49, 50]

Sony has not yet announced the price of the PSVR2 at the time of writing this thesis, but it is rumoured that it will be around the retail price of PS5, which based on the announced specifications would be a great bargain. The question is whether Sony will develop an official PC driver for it, or if there will be an unofficial one that will for example communicate with this headset according to the OpenXR standard. If so, the headset would become the cheapest VR headset with ET on the market capable of being used for ET research.

3.3 Existing solutions

This section summarises the available solutions that use a VR headset to collect and visualise ET data with the requirement of using game engines. Given how advanced modern ET hardware is, the number of software implementations that use it is scarce. The main player is Cognitive3D, which was the first to produce a robust solution that addresses this and does so using current game engines.

The Master's thesis written at the Faculty of Informatics of Masaryk University in 2020 explored the possibilities of ET technology in the context of VR, where the student designed and implemented a software solution to record, analyse, and visualise gaze data in Unity with the use of the HTC VIVE Pro Eye headset for VR [9]. The solution was used for a user study exploring escape behaviour. The student did not evaluate the gaze data in real time but first collected them into an external CSV file, which was later analysed. To visualise these data, he used visual indicators that were assembled from basic Unity primitives such as a cube, a sphere, or a lineRenderer. Points of fixation (PoF) are visualised in the thesis as concentrations of several ET points in a 3D space, which are coloured by a white-to-pink gradient that is supposed to liken them to heatmaps.

3.3.1 Cognitive3D

Cognitive3D is a company based in Vancouver, Canada, that offers an entire proprietary system made up of several parts for the collection and analysis of ET data. The system includes a streamlined production of use-case scenarios that are defined using objectives. The system can also be used for VR environments that do not employ ET in any way. It is applicable to academic and marketing research, VR training simulation, or architectural design. It has been designed to allow any content from Unreal or Unity to be integrated into the system using the corresponding *plug-in*. The *plug-in* takes control of the entire scene in an engine. The annual academic licence is in the range of tens of thousands of dollars. [51, 52]

Dashboard

The application that holds the whole platform together is called Dashboard. It is a web interface that contains several different functions and handles the structure of multiple users/organisations and their projects. [53]

A *customer* is an entity with a paid licence that can create various projects. Each *project* can have multiple defined *scenes*, which are containers for static scene geometry, *dynamic objects* and *session* data. A *session* is a specific instance of a single participant measurement that collects session data during the runtime of a Cognitive3D application that contains a scene of an experiment.

SceneExplorer

One of the main functions of the Dashboard is the SceneExplorer, which is used for a detailed review of the participant's session. This session contains a timeline that stores all actions, events, user movements and gazes, and fixations on objects performed during an experiment. The timeline can be stopped at any time to explore the state of the scene. It can display exactly where the participant was looking. The gaze is visualised as a'heatmap that fades according to a time fall-off. Fixation points are displayed using a trail consisting of spheres (fixations) and lines (saccades). [54]

Gaze and Fixations

Cognitive3D can be used without an ET headset, but in this case fixations and their metrics cannot be collected during the experiment, only gaze, which is collected based on the virtual position of the HMD. Gaze metrics are recorded always, but without an eye tracker the application assumes that the participant's focus is directly in front of the HDM. In this case, the metrics are significantly less accurate. [55]

Metric	Description
Average Gaze Count	The average number of times someone gazed at the object during one session.
Average Gaze Time	The average time the object was gazed at during one session.
Gaze Ratio	Reports the percentage of sessions in which the object was gazed at.
Total Sessions with Object	The number of times the object was present in a session.
Total User Gazes at Object	The total number of all gazes.
Sessions with Gaze	The number of sessions in which the participant noticed the object at least once.
User Gaze Length	The total duration of all gazes.
Gaze Instance Duration	The average duration of one instance of gaze.
Gaze Sequence	The sequence of all objects that were seen earlier.
Time to Gaze	The duration from the beginning of the session to the first gaze of the object.

Table 3.2 Cognitive3D gaze metrics of dynamic objects. [56]

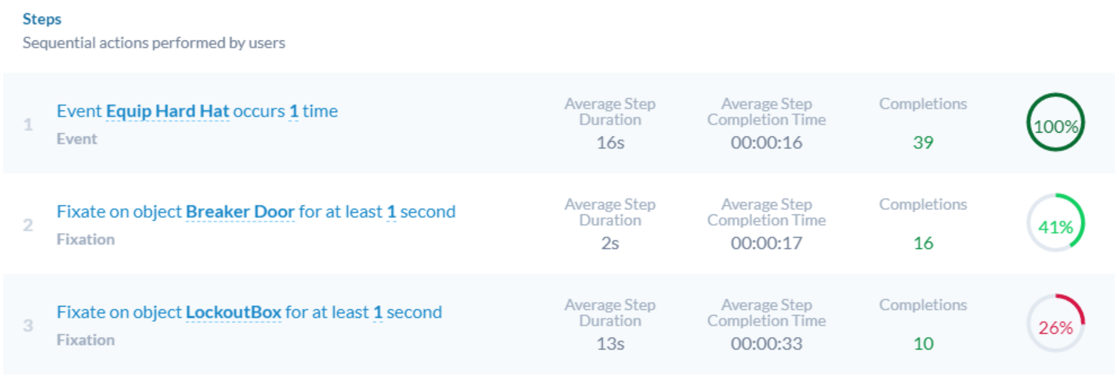
Metric	Description
Completions	The number of users that completed the step.
Completions in %	Total number of completions compared to the total amount of sessions.
Average Step Duration	The average time to complete the step relatively to the previously completed step.
Average Step Completion Time	The average time to complete the step relatively to the session start.
Successful Sessions	Total number of sessions that have cleared successfully every step.
Total Sessions	Total amount of sessions with the same objective.
Success Rate in %	The total number of successful sessions compared to total sessions.

Table 3.3 Cognitive3D metrics used for objectives and its steps. [56]

Gaze and fixation metrics are aggregated for individual dynamic objects that represent regions of interest (RoIs) in the scene. A group of several objects can be defined and treated as a separate RoI. Both metrics have the same structure, but their description is presented only on gaze metrics [15]. The Table 3.2 shows gaze metrics that are collected for each object independently over all sessions of a given scene. The last two metrics are collected only in the context of a single session. [56]

Objectives

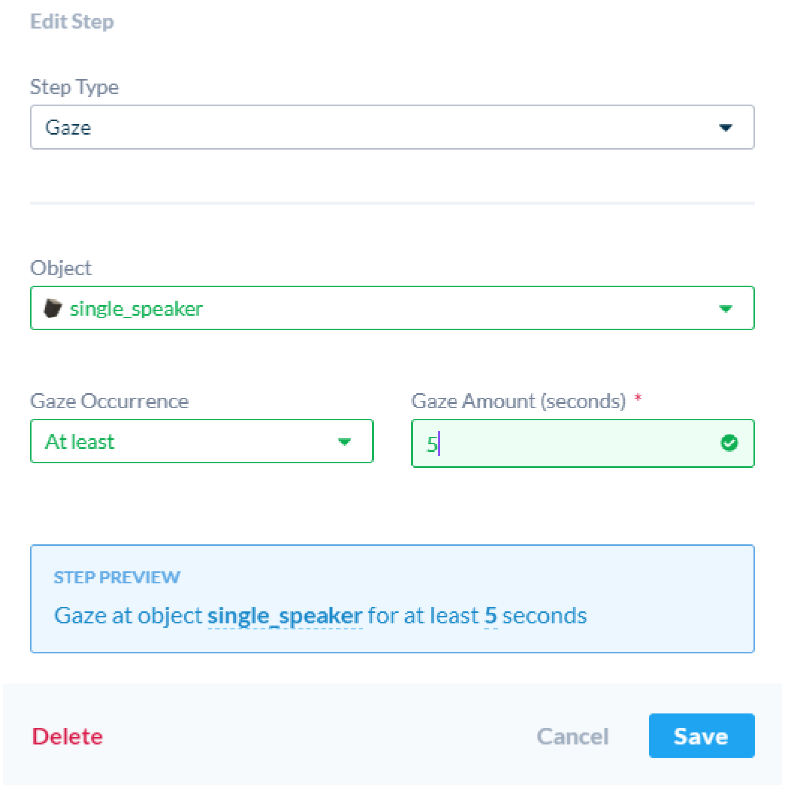
The next main functionality of the Cognitive3D software are the objectives, which are composed of individual steps and thus represent a certain script of what should occur during a session. They allow to collect information about what the participant was able to complete within a session or if they followed a certain process. Any amount of objectives can be defined in the context of a scene, and any amount of these objectives can be assigned to a session. An example of one with a list of individual steps and their calculated results in the Dashboard web interface can be seen in Figure 3.11, and Figure 3.12 displays the dialogue box in which the criteria of a single step for its completion are set. [57]



Steps
Sequential actions performed by users

		Average Step Duration	Average Step Completion Time	Completions	
1	Event **Equip Hard Hat** occurs **1** time Event	16s	00:00:16	39	100%
2	Fixate on object **Breaker Door** for at least **1** second Fixation	2s	00:00:17	16	41%
3	Fixate on object **LockoutBox** for at least **1** second Fixation	13s	00:00:33	10	26%

Figure 3.11 Sequence of steps and their completion in Cognitive3D. [57]


Edit Step

Step Type: Gaze

Object: single_speaker

Gaze Occurrence: At least

Gaze Amount (seconds) *: 5

STEP PREVIEW

Gaze at object single_speaker for at least 5 seconds

Figure 3.12 Setup of a step in Cognitive3D activating at gaze. [57]

Registering whether a step was successful can be easily evaluated using ET by whether the participant fixated or gazed at a particular object. A step of an objective is considered to be completed if it meets the specified criteria. It is possible to set the number of required fixations or their duration. Using this input, the first four metrics shown in Table 3.3 are collected for each step; the same metrics can be seen in Figure 3.11. The remaining three metrics of Table 3.3 are collected for the objective as a whole.

Supported hardware

The system is built to be compatible with a wide range of devices. It offers a high number of VR devices without ET. Among the VR HMDs with ET, it supports all of the previously mentioned with Tobii eye tracker in Section 3.2.2 and all Varjo headsets in Section 3.2.4. Support for XTALs did not exist during the writing of the thesis.

3.4 Unreal Engine capabilities

Unreal Engine (UE), developed by Epic Games [58], is used to render 3D computer graphics in real time. First designed for game development, it has found use in many other industries. Engine development is possible in C++ or using *Blueprint Visual Scripting*. This section first briefly introduces the tools used for development inside the engine [59] and the options on how to process gaze data for heatmap production in real time.

3.4.1 Blueprint Visual Scripting

Blueprint, short for *Blueprint class*, is based on a node-based interface to create, modify, and control elements in an Unreal scene. It is essentially a class that is stored as an Asset in the Content Browser. It allows designers to use programming tools without having to touch the C++ language.

Any functionality that Blueprints do not offer can be programmed using C++ and then made available to Blueprint nodes. A Blueprint does not have to contain a functionality for a specific gameplay element. It can exist fully independently as a collection of static functions. It is called *Blueprint Function Library*. Unreal Engine already comes with a rich set of tools that define many Blueprint classes and libraries. Many Blueprint classes can be used as a parent class for new ones that inherit their functionality.

3.4.2 Tools and Classes

For this thesis, it is crucial to know these UE options. The scale of an UE scene is defined in centimetres.

Asset is a piece of content, the engine can work with, that is located in Content Browser; part of Unreal's interface. Assets are files stored with `.uasset` extension. They represent, for instance, imported models and textures, or Blueprints.

Actor is a class that by default maintains 3D spatial information about its position and rotation, which can always be modified. An Actor is spawnable and destroyable at any time. It can be, for example, a controlled character, cameras, lights, or other 3D models. Each Actor in a scene has its own Blueprint and a *Component graph*; hierarchical tree of Components.

Pawn is an Actor that is the essential physical representation of a player within the world. It is the base class for all Actors that can be controlled either by a player or an AI. It is possible to add a DefaultPawnMovementComponent to a pawn to extend it with simple no-gravity movement.

Character is a special type of Pawn that offers basic movement abilities for vertically orientated characters that can walk, run and jump.

Component is a Blueprint class that provides functionality that is attachable to any Actor at any time.

Level is a special type of Asset that serves as a hierarchical collection of several different Actors to create a scene. Each Level in a project has one Blueprint assigned by default.

Tick is defined within the engine as a regular interval that usually occurs once every frame. An actor may have a defined OnTick function that is repeated periodically. It is often used to update a given Actor.

Material is an Asset that defines how a surface should be rendered. It is defined using visual scripting nodes – Material Expressions – in the Material Editor, which is not a Blueprint. Each material is translated as an HLSL programme executed on a GPU consisting of a vertex and a pixel shader. The nodes replace pieces of HLSL code.

Static mesh is an object class that represents the fundamental renderable static geometry. It is generated by importing a 3D model into a UE project. It can be edited in the Static Mesh Editor, which includes, among other functions, collision, UV map, and LOD settings or Material assignment. A static mesh also exists as a Component or an Actor, which effectively contains a StaticMeshComponent as the root of its graph. The mesh is rendered according to the assigned material.

Material Instancing is a method that is used to change the appearance of material without recompiling the HLSL programme. Some Material Expression nodes such as vectors, textures and floats can be turned into parameters that are set before the Material is rendered. This requires to construct a Dynamic Material Instance of the given Material.

Canvas Render Target 2D is a Blueprint object class that inherits Texture class. This is a texture that can be constructed, modified, and exported at runtime.

Scene Capture 2D is a camera Component that captures a scene from its position using its view frustum into a texture. It can record the scene continuously every Tick or just once by calling its Blueprint function *Capture Scene*.

ShowOnly List is an option inside the Scene Capture Component's settings that allows to render only those objects that are in the list. This option must be explicitly enabled first.

Line Trace is a Blueprint function, callable only from an Actor, that will perform trace collisions along a specified 3D line; parameters Start and End. The function returns the first object hit by this line.

Plug-in is a collection of code or data that developers can enable or disable within their projects. Additional plug-ins are installed by placing them in the project's Plugins directory or the UE root directory in the [Root]/Engine/Plugins folder.

3.4.3 OpenXR

There are different ways to interact with VR, XR, and AR devices in Unreal. However, it offers a unified way to access them using OpenXR. To activate certain features, it is crucial to enable a certain plug-in. Basic compatibility is guaranteed by the OpenXR plug-in. Additional functionality can be activated by enabling OpenXRHandTracking and OpenXREyeTracker.

Due to potential conflicts and errors, it is recommended to disable other proprietary plug-ins designed to communicate with their hardware and other methods such as SteamVR, OculusVR, or Oculus OpenXR.

Unreal offers a simple interface for ET, through which gaze data from OpenXR are sent when the plug-in is enabled. *Get Gaze Data* returns a combined gaze, *Get Stereo Gaze Data* returns two gazes, one for each eye. The gaze data structure consists of an origin and direction vector in the absolute real world coordinates of a tracking system used with the headset.

3.4.4 UV mapping

A static mesh is a geometric shape that remains unchanged and always contains the same number of vertices and faces. The way static meshes are rendered depends on their attached Material. Often these meshes are rendered with a texture according to their UV maps.

UV mapping is a procedure in which a 2D image is projected onto a 3D model. For this to happen, it is necessary that the 3D model has its own UV map, which assigns to its vertices its own (u, v) coordinate where $u, v \in [0, 1]$. A texture or an image has its width and height scaled to the range of the UV map during rendering. The areas formed by connecting the (u, v) coordinates of the vertices project the corresponding piece of a texture onto the 3D mesh itself.

3.4.5 Mesh painting

Unreal offers the ability to create dynamic textures using Render Targets and Dynamic Materials. Drawing on the mesh itself can be simulated by modifying its texture. For this to work, the UV map must not have overlapping faces defined. Each face of the 3D model must have a uniquely assigned area in the texture. This is also required by lightmaps in UE, which are used to bake scene lights into textures themselves, so they do not have to be computed during rendering.

In the context of ET data visualisation, there is the question of using mesh painting to produce heatmap textures based on a gaze vector that collides with the given object. Render Target can be exported in runtime.

It needs to be tested if this method of data collection is worth doing in real-time performance-wise, and how much storage it will require with multiple objects in the scene – each must have its own heatmap texture.

3.4.6 UV painting method

The first approach to drawing heatmaps on textures would be to use a collision hit position of the object, which will be mapped to the appropriate (u, v) coordinate. This requires enabling the Support UV From Hit Results option in the Engine – Physics section of the UE project settings.

This Unreal tutorial [60] shows a method for painting with a brush defined using a Material, which is set to additive blend mode. This is necessary to add new values to the resulting Render Target instead of replacing them.

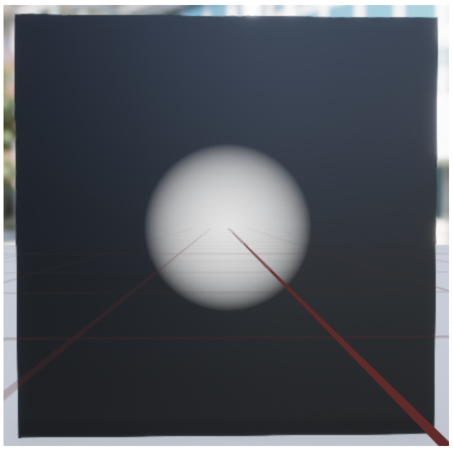


Figure 3.13 Preview of 2D brush Unreal Material. [61]

The brush material is shown in Figure 3.13. The appearance of the preview corresponds to the brush settings of 0.2 radius and $(0.5, 0.5)$ coordinate on an UV map. The material has these values as input parameters that can be set before rendering to a Render Target. As these parameters change, the brush will change its position and size.

Render Target must be a square texture that directly shadows the UV map range. This texture is set to a 16-bit colour depth for each channel for greater detail of the brush.

It is beneficial to first record the heatmap as a brush trail for easier manipulation of the visualisation. One material can be seen in this Unreal tutorial [61]. The recorded trail can be displayed as a heatmap on a model using additional material that will be responsible for its appearance. The parameters of the Material are set by its Material Blueprint functions with the wording *Set Parameter Value*. A Material is rendered to a Render Target using the Blueprint function *Draw Material To Render Target*.

Problems

There is however a major drawback to this method. Brush splats are always drawn directly on a texture. It does not reflect the actual geometry and size of the 3D model in the scene. The UV map is not guaranteed to have all faces defined on the same scale or aspect ratio. For such models, this would cause the heatmap trail to be stretched for some faces or to be of a different size.

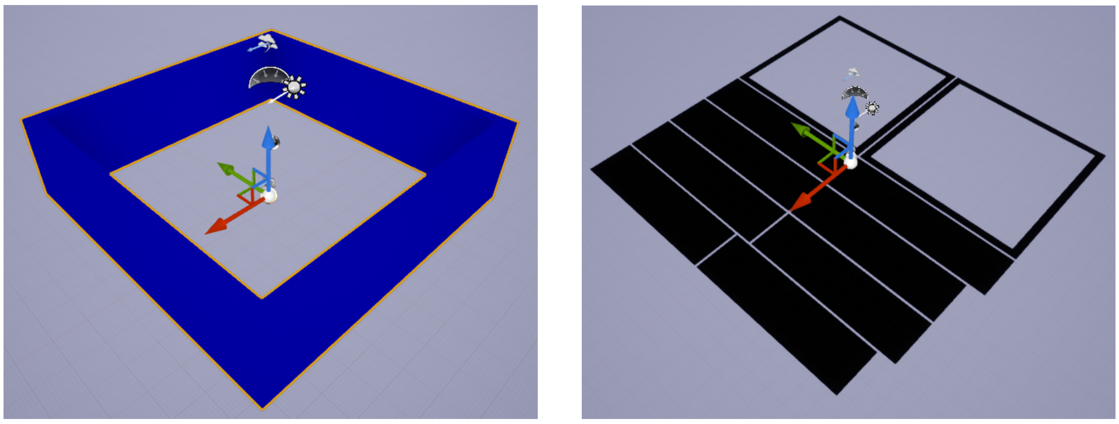
For symmetric, uniform, and unambiguous models with a clearly defined UV map, such as a cube or a plane, this will certainly work much better. But it is also inadequate because it does not prevent brush overflow, where the drawing stroke on a certain face overflows into another. An example of this issue is demonstrated in the `video/01-basic-painting-method.mp4` in the enclosed DVD. The wall model in the video is a single object. This solution is unsatisfactory and a more suitable one is needed to address these problems.

3.4.7 UV unwrap method

In this blog post [62], Unreal Engine veteran Ryan Brucks discusses automatic UV dilation to prevent edge bleeding for lower resolution mipmaps. In the process, he also published his method for unwrapping meshes on the basis of its UV map. The first step in achieving dilation involves unwrapping the model in the world according to UV. The object appears in the scene as a 3D plane, which is rendered using an orthographic Scene Capture Component to a Render Target with a 32-bit pixel depth. In this texture, 3D coordinates of the mesh's absolute position in the world are captured.

Dilation is then achieved by stretching the faces' edges on the UV map using algorithms that make use of other UE Materials. What is needed mainly for this thesis is the unwrap method. Use a brush that imprints itself on the texture in the exact areas around where a gaze vector has hit the object.

This requires using the absolute world coordinates of the mesh vertices while arranging the vertices of an object to appear on a single unwrap plane that will be recorded in a texture. The way a model looks when it is unwrapped can be seen in Figure 3.14, and how it is implemented in the Material Expression nodes is shown in Figure 3.15.



(a) Wall mesh. **(b)** Unwrapped wall mesh.

Figure 3.14 Difference between applied unwrap material.

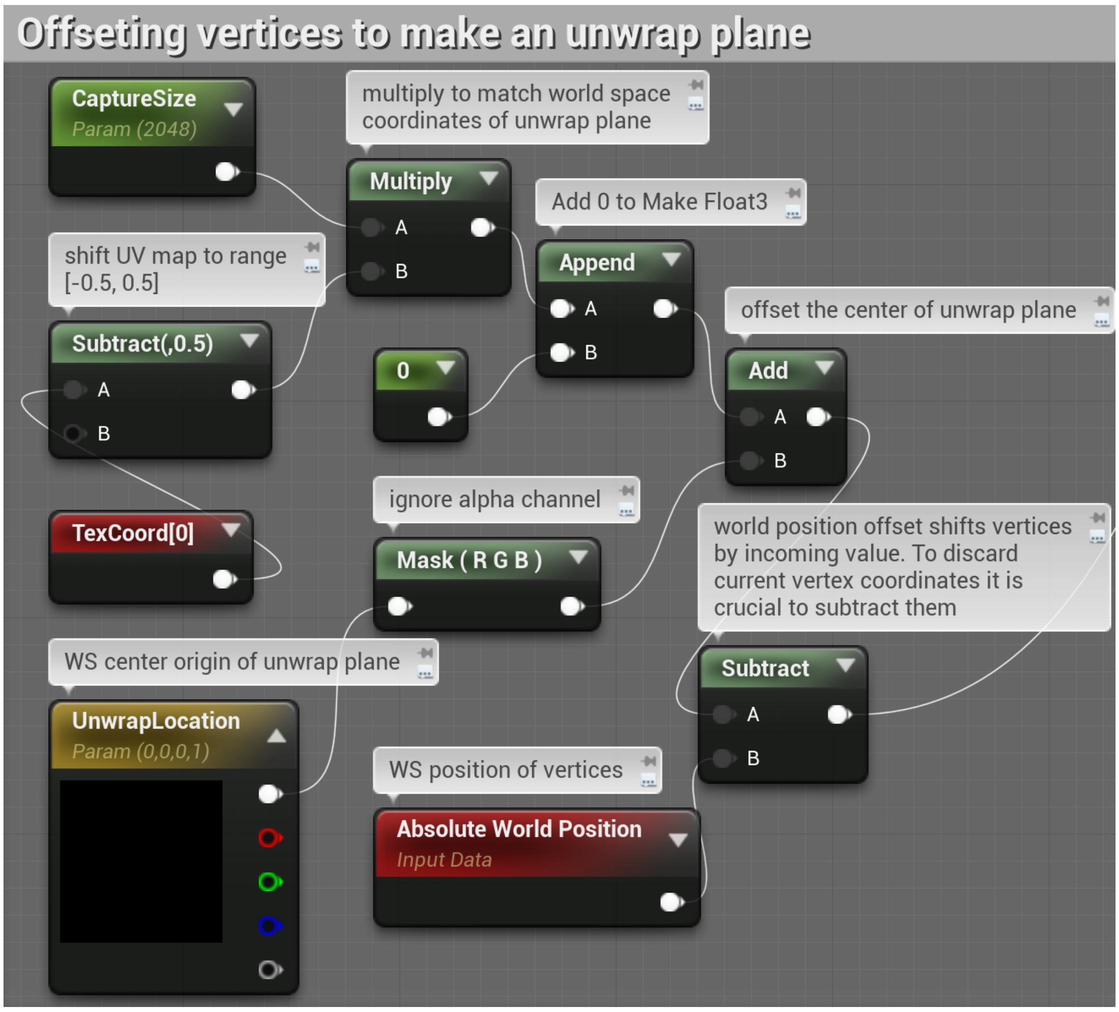

Offseting vertices to make an unwrap plane
CaptureSize
Param (2048)
multiply to match world space coordinates of unwrap plane
Multiply
A
B
Add 0 to Make Float3
Append
A
B
shift UV map to range [-0.5, 0.5]
Subtract(,0.5)
A
B
0
offset the center of unwrap plane
Add
A
B
TexCoord[0]
ignore alpha channel
Mask (R G B)
world position offset shifts vertices by incoming value. To discard current vertex coordinates it is crucial to subtract them
WS center origin of unwrap plane
Subtract
A
B
UnwrapLocation
Param (0,0,0,1)
WS position of vertices
Absolute World Position
Input Data

Figure 3.15 Material Expressions used to unwrap mesh to world space. [62]

The result of this algorithm is a 3D vector that is plugged into the *World Position Offset* pin in Material's result node, which is added to the total World Position value of a vertex. The algorithm consists of creating a plane in world coordinates from which the absolute world position is subtracted, so that the resulting shape is the desired unwrap plane.

CaptureSize parameter specifies the size of the square texture in which it will be rendered. For a 2048px texture, a plane of size 20.48 metres will appear in the world, because the scale of Unreal is in centimetres. This must be set to always add the correct values to the texture.

UnwrapLocation vector parameter moves the centre of the unwrap plane in the world. Objects can reside at different locations in the scene, and hard assigning values to the scene's origin can cause problems. It may happen that object values are going to be recorded in an unwrapped state and nothing will be added to the texture. This is simply caused by the object having been culled during the rendering. The actual position of the object is somewhere else in the scene, outside of a view frustum range.

Modification

Tran proposes a modification to this method [63] by not storing the position in an assistive Render Target, but creating a 3D brush in the same Material that essentially does the same thing as the simpler method. Instead of comparing the (u, v) coordinate of the hit location and the (u, v) coordinate of a pixel, it compares the hit location to the pixel's absolute world position without Material offsets applied. The implementation can be seen in Figure 3.16 which computes the resulting colour that is supplied to the Emissive Colour of the Unwrap Material.

BrushSize parameter specifies the size of the 3D brush radius. *Strength* parameter determines how hard the brush imprints on a texture. It must be in the range [0, 1]. A value of 1 would create the maximum value on the heatmap and 0 would not make any changes.

Now the object is coloured based on a 3D brush that takes into account the absolute position of the mesh in the scene. This method eliminates all the problems that the simpler method had.

Tran's tutorial also offers a basic Blueprint setup for recording this UV unwrap material using a Scene Capture Component and a black projection plane. For more information on the implementation, see Section 5.2.2

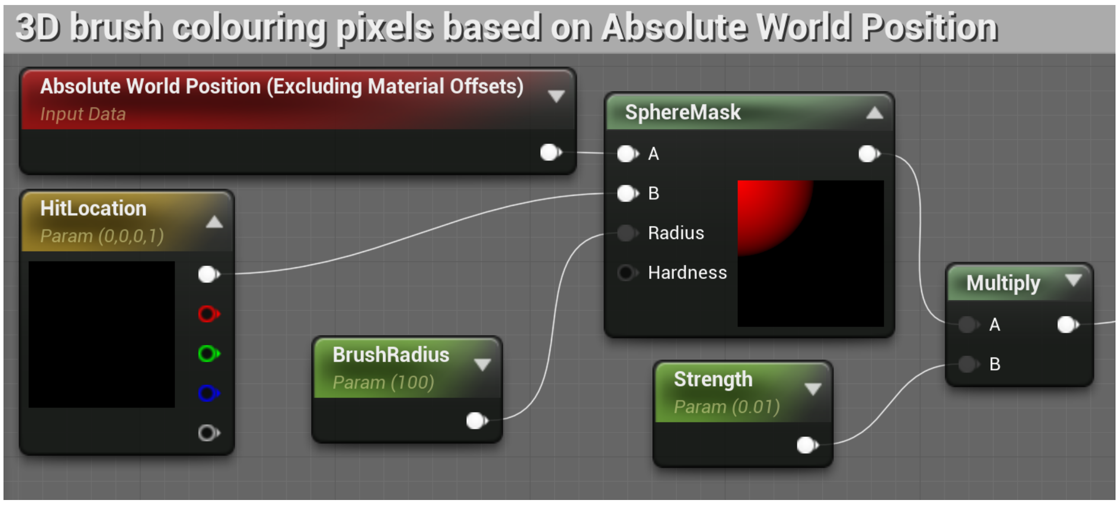


Figure 3.16 3D Brush defined in Material Expressions.

3.4.8 Runtime texture loading

Exporting a Render Target is simple, all it requires is a single function called *ExportToDisk*. The function takes a Texture Object reference, which is the parent of the Render Target, a path with a filename, and export options. Importing a texture directly into the Render Target is more complicated, as no such function exists. It is necessary to obtain the assistance of a material shown in Figure 3.18. The process is shown in this tutorial [64].

Texture loading options exist, but only as a Texture 2D object. This is done by the *Import File as Texture 2D* function, which takes a file path as input. The whole method is to redraw the loaded texture into a Render Target. Load Material does exactly that; it just sets the texture parameter itself as the output colour of the material. This parameter must be set from the Dynamic Material Instance by calling its Blueprint function Set Texture Parameter Value – Figure 3.17. The input value for this function is the loaded texture, and the Parameter Name. That must be the same as the Param2D name in the Material.

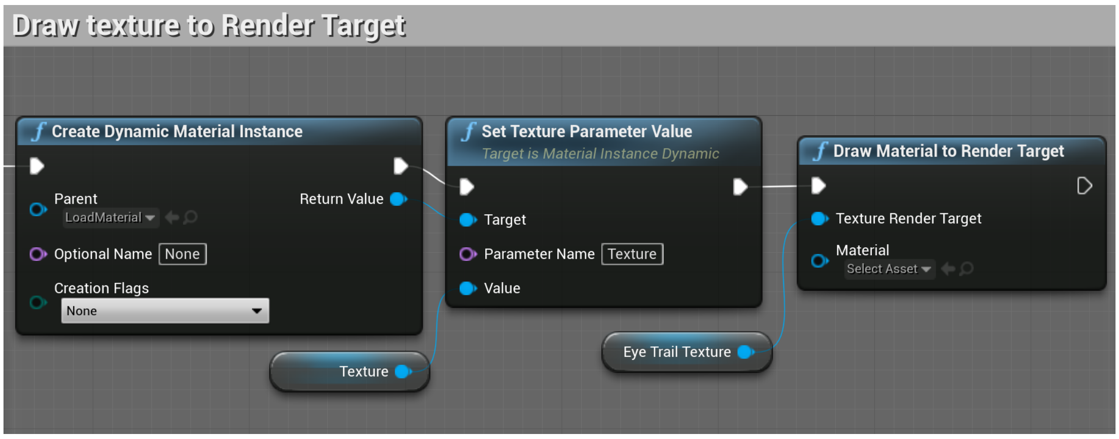


Figure 3.17 Process of importing texture on Render Target.

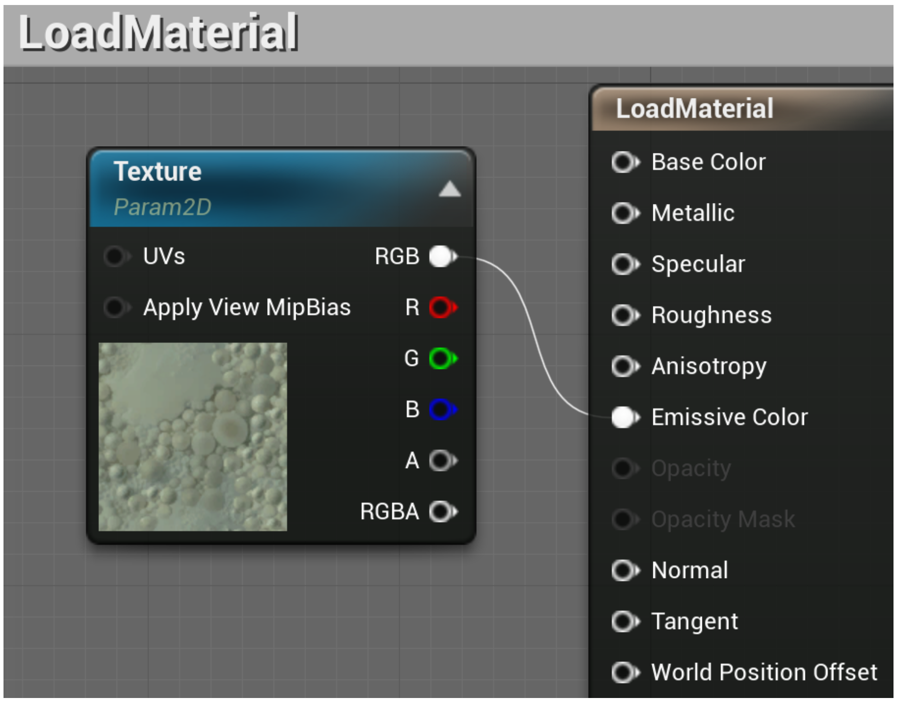


Figure 3.18 Unreal Material used for loading. [64]

Chapter 4

Design

The design of the application prototype is being covered by this chapter. The form of the prototype is chosen and a specific solution is proposed.

4.1 Requirements

The thesis assignment and the consultations with the supervisor provide functional and non-functional requirements of the prototype.

4.1.1 Functional requirements

Functional requirements define what functionality the prototype should contain. While creating the assignment, it was unclear what form it would assume and what methods would be used.

Gaze data collection The prototype application should be able to collect gaze data in some way provided by a VR HMD with ET.

Gaze data visualisation The prototype shall choose an appropriate visualisation technique that will be applied to the collected gaze data.

Environment for data collection and visualisation A test scene, or experiment, in which gaze data can be visualised and collected from users moving through the scene.

Make resultant data Data are to be collected from multiple users in a same environment and combined into resultant data.

4.1.2 Non-functional requirements

Non-functional requirements demand certain qualities and define the limitations of the prototype.

Technologies The main functionalities of the prototype such as data collection and visualisation, must be developed using a game engine of choice, Unreal Engine or Unity. Standalone supporting programmes written in *C++* or *Python* can be used to achieve some of its requirements.

Documentation All code must be commented on and described in good detail.

4.2 Prototype application

This section covers the thought process considering the situation in which the prototype was designed, and results in a decision on what technologies will be used and what form the prototype application will take.

4.2.1 Selected technologies

Software

An analysis of current solutions was undertaken – Section 3.3 – to select the most appropriate technology to allow for maximum opportunity to offer a new solution. The Masaryk University thesis dealt with a similar topic in the Unity environment. Cognitive3D offers a very robust, but proprietary, system that uses both engines. Based on this information, this thesis will focus on development in Unreal Engine, as no other open source solution to this ET problem has been found. Unreal version 4.27 will be used to develop the main functionalities of the prototype. Unreal Engine 5 had not been fully released when this decision was made. During the writing of the thesis, this was no longer the case.

Furthermore, the Python programming language with the Pillow library will be used to create one simple support programme that will process the collected gaze data.

Hardware

In terms of hardware, there were not many options available and it was generally difficult to find access to a VR headset with ET. There was an opportunity to use the 3D CAVE lab at the Faculty of Engineering at CTU, which contains an XTAL headset – Section 3.2.5.

Conveniently, there is no solution available that utilises the ET capabilities of XTAL and, therefore, will be used for this thesis. Furthermore, the device is of Czech production with local support, and it will be possible to discuss possible problems and issues with Vrgineers in a relatively short time.

4.2.2 Form

The requirement for the prototype refers to the technology used, but does not mention what form it should take. In order to select the best form of the prototype, the following situation had to be taken into account.

Situation

The XTAL headset was chosen for development, but was not physically available for testing at the time. Based on the analysis in Section 3.2.5, it operates on its native runtime and it was assumed, due to the lack of official data on its ET, that it would return gaze data in the structure from Section 3.1.2. When the headset became available, it was discovered that this was not true and the method of data collection had to be consulted with Vrgineers support, from whom a beta version of their OpenXR runtime was received. This did not happen until the later stages of development.

To avoid unnecessary time loss waiting for the headset's availability and the later consultations with support, it was decided to take the prototype concept a bit differently.

Proposal

The prototype application must be developed without the access to a headset in such a way that the collected data from it can be easily used when available. The task is to assume that gaze data are available and can be processed, but it is necessary to provide some alternative to substitute for their presence. The forward vector of the virtual camera inside the Unreal Engine scene can be used for this purpose.

This made it possible to think about conceiving the prototype as an Unreal Engine *plug-in* that will extend the functionality of the game engine or the Unreal project by the possibility of collecting and visualising gaze data. The plug-in will be independent of any ET hardware and will only require the origin and direction vector as input.

The plug-in has the advantage that it can be mounted on any UE project of the same version in which it was built. The assignment needs to be slightly adjusted to reflect this change.

4.2.3 Modification

The plug-in will cover the functionality of data collection and visualisation, but it still remains to create an experiment, the environment where the data is collected. The experiment will be done separately as an Unreal Engine project, and the plug-in will be used to build it. The plug-in must include functionality that will support the construction of the test environment.

The practical portion of this thesis is divided into two chapters. The implementation chapter will present the implementation of the plug-in itself, and the experiment chapter will describe the construction process of a test environment using the plug-in. It will also cover the use of the XTAL headset and the collection of data from multiple users.

The merging of the data of multiple participants will not be handled by the plug-in but rather by a simple Python script that will be run once after all the data have been collected.

4.3 Functionality

Based on the analysis, it was found that all available solutions record the data first offline and then process them.

This work will focus on a real time production of heatmaps that can be used both as a data collection method and as a data visualisation. Heatmaps will be produced using the UV unwrap method utilising Unreal Materials as described in Section 3.4.7. Heatmap textures are going to be loadable and exportable.

Two heatmaps will be recorded at the same time. One will typically be created using the eye gaze, but the other one will be created based on the forward vector of a virtual camera inside the scene. Both heatmaps will be produced at the same time for later comparison. This test is intended to be a direct demonstration of the usability of ET. The collected texture with a heatmap trail will be displayed on its object using Material. There will be another Material that will directly visualise their mutual appearance.

This thesis will attempt to implement the entire prototype using only the built-in visual programming via Blueprints in Unreal Engine. This is meant to serve as proof that the current tools are advanced and that there is really no reason why eye tracking in Unreal Engine should not be pursued by other future projects.

4.3.1 Actions

When designing a plug-in, it is worth imagining how an experiment employing it will look like. The activity diagram in Figure 4.1 describes exactly one of these experiments. It is a simple one that only initiates, runs, and ends after a certain amount of time. Once it starts, a game cycle is triggered that sends two rays into the scene with each Tick. These, when they collide with an object that has a heatmap, will add a value to the heatmap texture.

This leads to four important actions that the plug-in must take into account. The first is the functionality of shooting rays and processing them. Second, the colliding ray must recognise if it is an object that has a heatmap or if it is an ordinary scene object. Third, imprinting a value on one of the object's textures. The object will have two of those, so the colliding ray classification needs to be added. And finally, export all heatmaps.

Eye Tracked object

This is an object that will be modifiable with this plug-in. It must be defined as such to be affected by a ray. In the context of the Unreal Engine, it should be a Component or an Actor that has direct access to static mesh data.

During the creation of an experiment, the desired functionality is that it should be sufficient to insert this type of object into the scene, not worry about much else, and be assured that a ray will collide with it. As part of the plug-in design, this object can also be used for possible subsequent iterations for other ways of visualising or collecting data.

The object will contain an export function that saves both textures to a local drive. In order to save all these textures of all objects, a static function will have to be defined to iterate through all the Eye Tracked objects in the scene and use their respective export function.

Raycaster

This object will be responsible for sending rays into the scene based on the input of the origin and direction vector. Within a test scene, these rays should be emitted from a defined Pawn or Character that is used to navigate through the scene. It is logical to use the collision function from this perspective so that the function itself does not use this Actor during collision calculations. If an Actor unrelated to the emitted ray were used for these, the collision would never reach the world because it could be trapped inside a collision volume of the Actor, which would be part of the collision calculations, unless excluded.

One could certainly create such a Pawn within a plug-in and hard-code the raycasting capabilities into it, but this is not a solution that is extensible, as each scene may require different scene traversal logic and functionalities.

Therefore, raycasting must be implemented within a Component that can then be added to any Pawn or Character.

Heatmap renderer

The heatmap rendering method will be adopted from this article [63] with a static mesh UV unwrap Material defined in Section 3.4.7. The entire rendering depends on an Actor that must be in the scene. This Actor has to contain a black projection plane placed on the floor of the scene that is the size of the unwrapped mesh, i.e. for a 1024x1024px texture, a square plane with an edge length of 100 cm has to be scaled 10.24 times. The plane creates a black background so that the renderer avoids parasitic surrounding effects.

The Component that will capture this is the Scene Capture Component, which will have a defined Show-Only List that will contain only the rendered object with a black background; only those two will be rendered. It must be rendered in orthographic mode, where its view width must be equal to the width of the texture that will be modified.

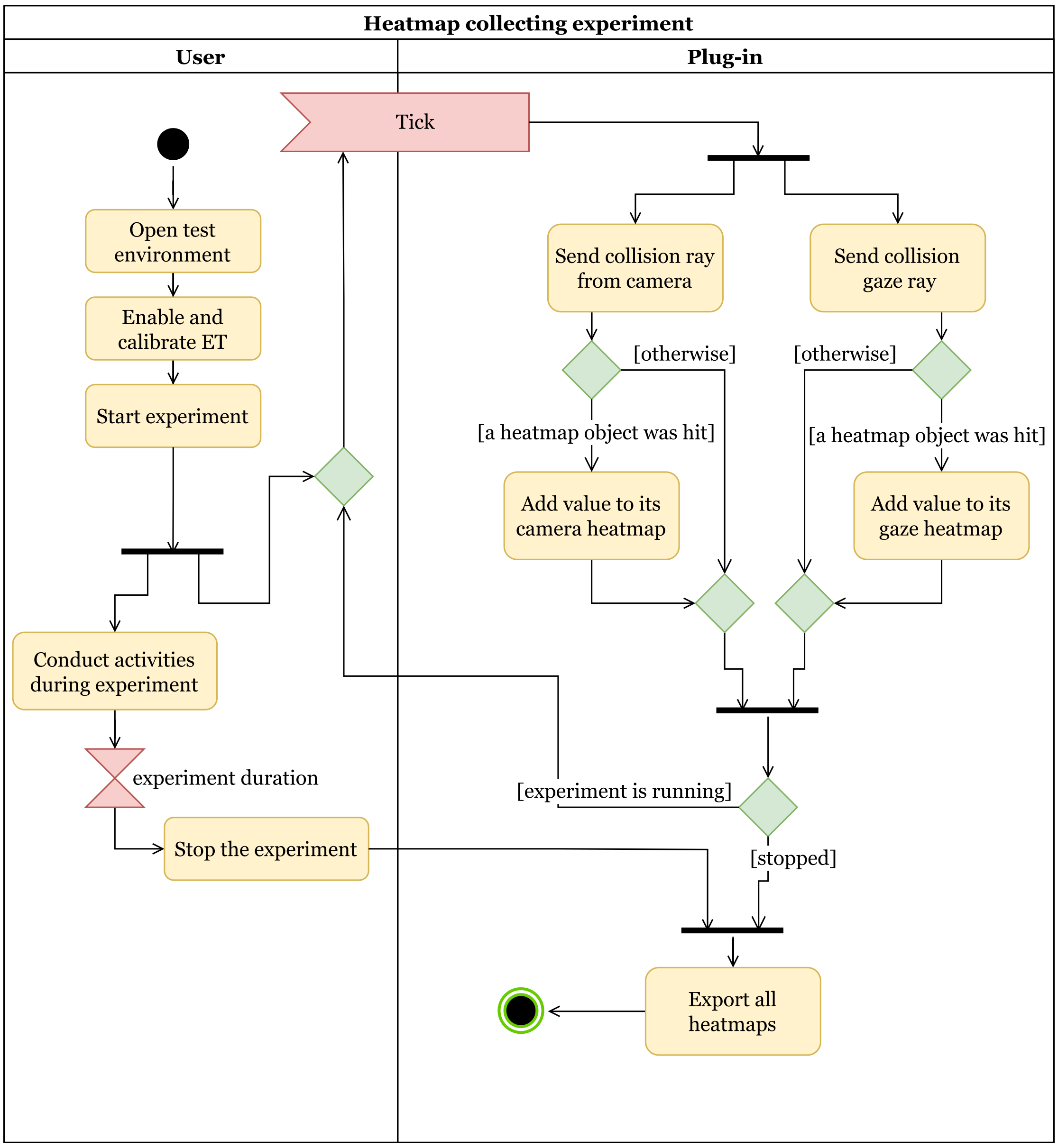


Figure 4.1 Activity diagram of a possible ET experiment collecting heatmaps.

Chapter 5
Implementation

The implementation of the EyeTrackingUtilities plug-in is described in this chapter. When added to an Unreal Engine project, it offers a Blueprint and Material directory in its Content folder. The project can immediately use its contents.

5.1 Materials

Materials are key to the functionality of visualisation and collection of heatmaps. All of them will be described in this section. Material Content folder can be seen in Figure 5.1.

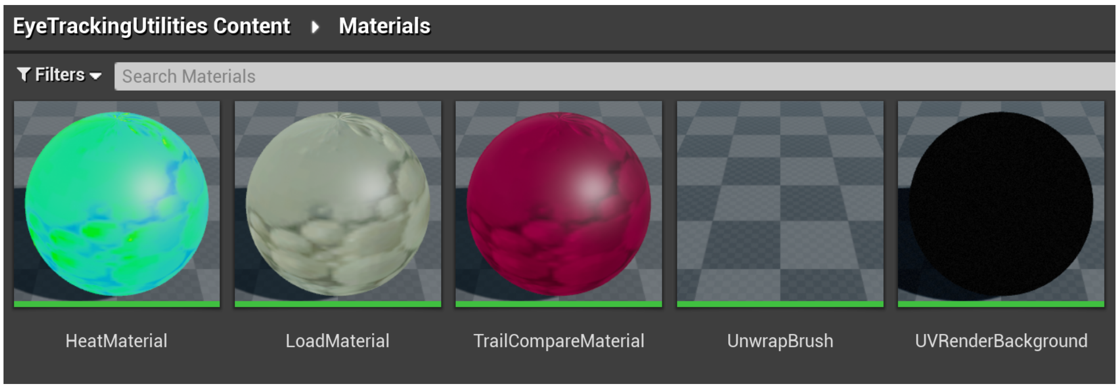


Figure 5.1 Contents of EyeTrackingUtilities Material folder.

5.1.1 LoadMaterial and UVRenderBackground

These are the two simplest materials. *UVRenderBackground* is a material that constantly assigns a black colour – a value of (0, 0, 0) – to the base colour, metallic, and specular pins of its result node.

LoadMaterial is exactly the same Material described in Section 3.4.8, to load already exported heatmap textures.

5.1.2 UnwrapBrush

UnwrapBrush is an implemented version of the Material from Section 3.4.7 using Tran's modification to skip several rendering steps using a 3D brush. The block of material expressions in Figure 3.15 is connected to the World Position Offset pin of the resulting Material node, and another part in Figure 3.16 is connected to Emissive Colour pin.

This technique enables rendering heatmap textures on a GPU, so the computation is very fast and does not slow down the overall performance even when rendering 2K textures. However, it does have one problem, which is related to the use of a 3D brush. If the brush is large enough or even larger than the object itself, it paints the parts of the object that are not visible. An illustrative video demonstration is available in the enclosed media in `video/02-3D-brush-problem.mp4`.

This problem was solved by adding two parameters related to a collision in the scene. The solution can be seen in Figure 4.2.

HitNormal parameter

HitNormal is the normal of the static mesh at the collision point in world coordinates. The reason to use this is so that one can determine which pixel to colour based on the size of an angle between the normal of a currently drawn pixel in world coordinates and the normal of the collision point. This is to prevent the brush from drawing on faces with an opposite normal; those that are at an obtuse angle to the collision point.

When this angle is greater than 92°, the pixel is assigned a black colour. A 90° could have been used, but that would have prohibited the simultaneous drawing on two perpendicular faces, which is desirable in some situations because both faces can be viewed at 45°.

But that does not solve the whole problem because it is calculated only from the object's perspective. It is still possible to draw on surfaces perpendicular to those on which the collisions occur, even though they are not visible from the camera view.

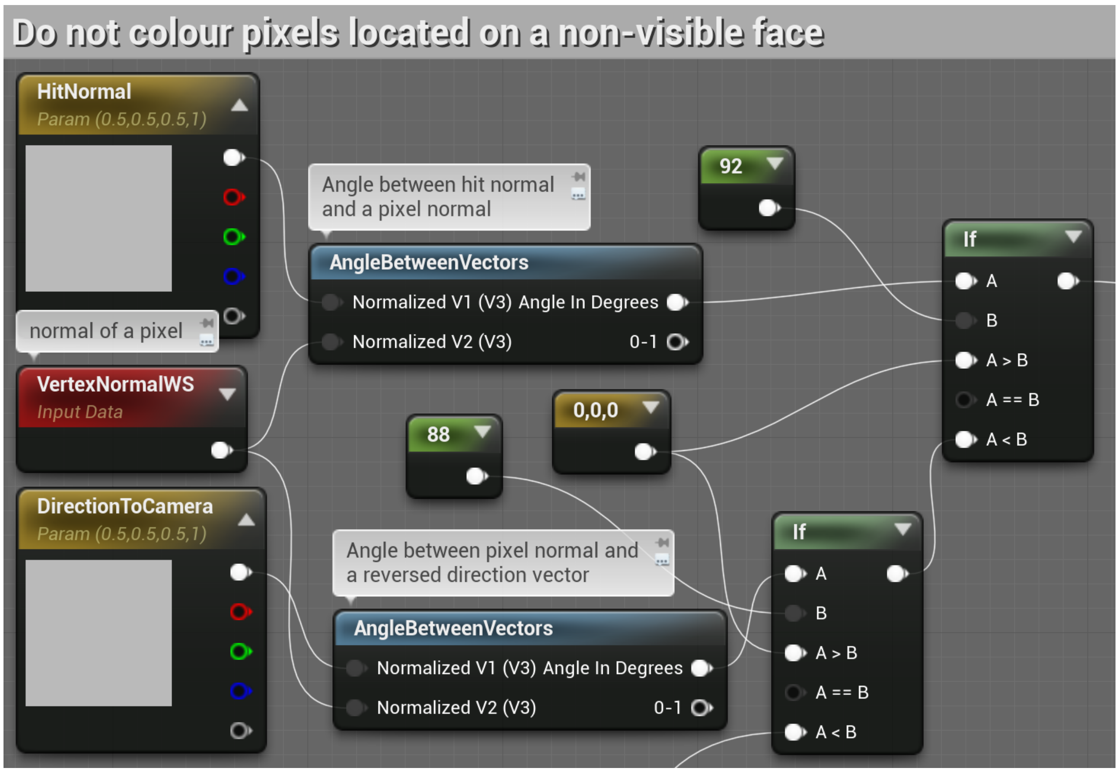
DirectionToCamera parameter

The *DirectionToCamera* parameter handles just that. The method is exactly the same as for the previous parameter, but a different threshold is set. It takes a direction vector to the camera and a normal of the pixel in the WS. It will draw on all surfaces as long as the camera is looking at them at 88°.

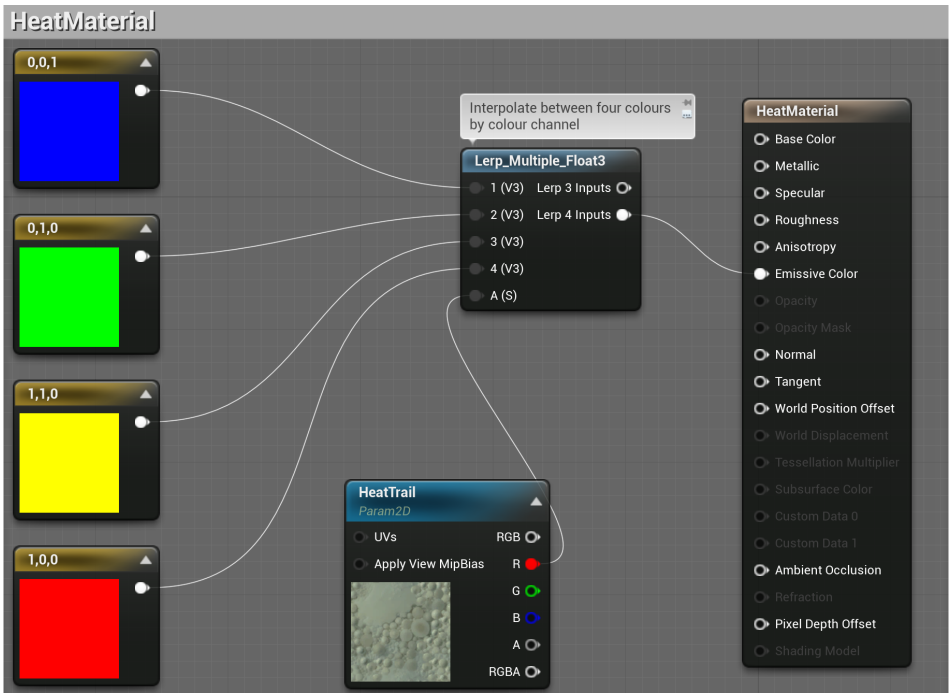
Note that using both of these angular conditions simultaneously will sometimes not guarantee drawing on a face that is visible but not colliding with a ray. The former condition will guarantee that one can draw only on one of the two faces that form an acute-angle edge. At the same time, however, it is not enough to use only the latter condition, because sometimes the brush may bleed through to the other side of narrow objects and acute angles of the camera.

5.1.3 HeatMaterial

Basic Material that visualises a heatmap texture on a mesh. Composed of four colours that are interpolated by the value of one texture channel, as shown in Figure 5.3 [61]. The result is projected as the colour of the mesh. A heatmap texture is added using the texture parameter *HeatTrail*.



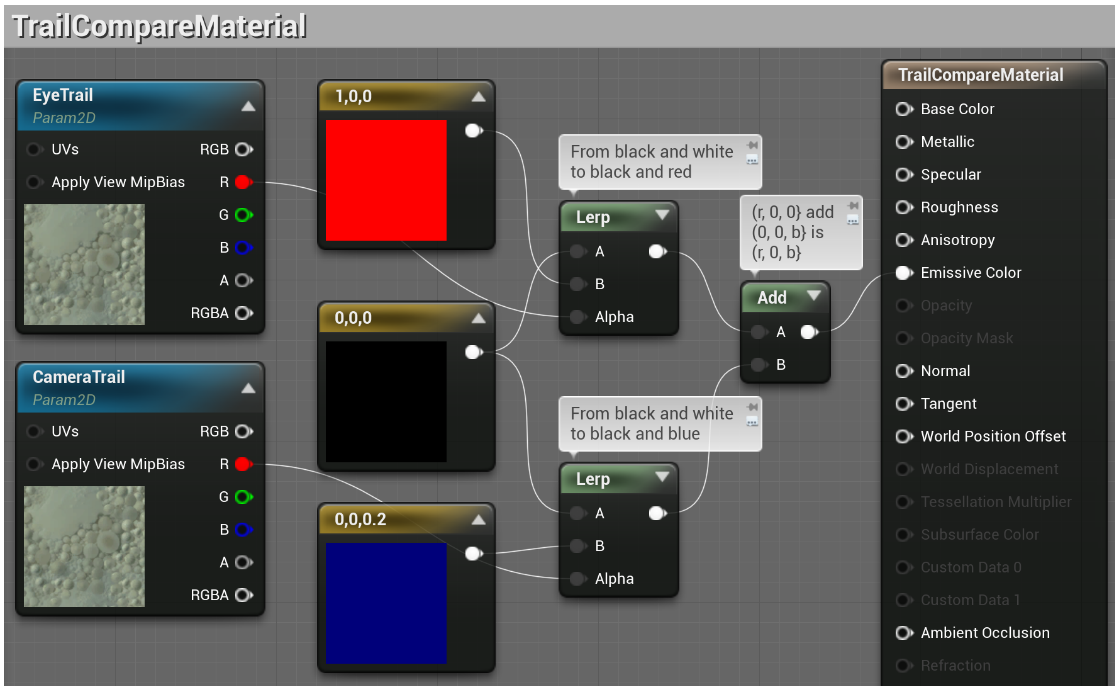
■ **Figure 5.2** Unwrap 3D brush material correction with angular conditions.



■ **Figure 5.3** HeatMaterial node definition.

5.1.4 TrailCompareMaterial

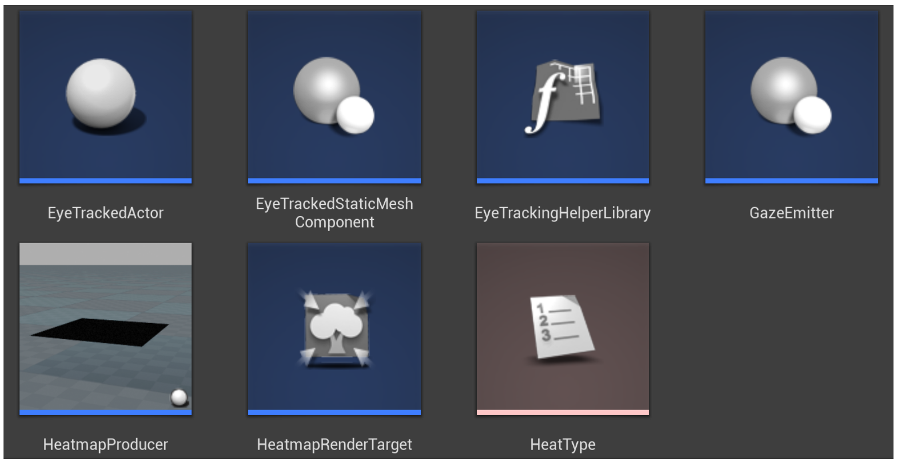
Simple material that combines two colours into one to visualise the appearance of two different textures on an object with a heatmap. The definition is described in Figure 5.4.



■ **Figure 5.4** TrailCompareMaterial node definition.

5.2 Blueprints

This section presents the implementation of classes with functionalities designed in Section 4.3. Contents of the plug-in's Blueprint folder is in Figure 5.5



■ **Figure 5.5** Contents of EyeTrackingUtilities Blueprint folder.

5.2.1 EyeTrackedStaticMeshComponent

EyeTrackedStaticComponent is a component inherited from *StaticMeshComponent*. This class extends the original static mesh functionality with two private render targets; *EyeTrailTexture*, *CameraTrailTexture*; and two private material instances; *HeatmapMaterial*, *CompareTrailMaterial*.

Both render targets have an adjustable resolution in one variable that is initiated once when the application starts. One is enough, because the render targets must be a square. Furthermore, it can be decided that a given component can be EyeTracked but does not have to have its own heatmap, so a public boolean *AcceptsHeat* property was added.

The render target is initialised by calling a single Blueprint function called Create Canvas Render Target 2D, which is given a width and height. This is done twice in total, as illustrated in Figure 5.7. Furthermore, both instances of dynamic materials are initialised. Blueprint nodes of this operation can be seen in Appendix B.

Render targets are private variables. To use them for rendering, a Scene Capture Component from other classes will have targets assigned within this class. The function *AttachRenderer* in Figure 5.6 serves this purpouse.

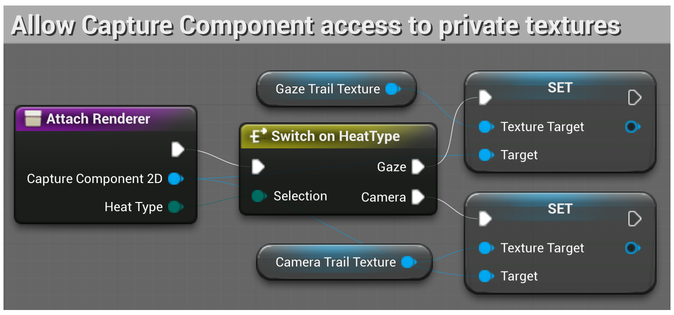


Figure 5.6 AttachRenderer function in EyeTrackedStaticMeshComponent.

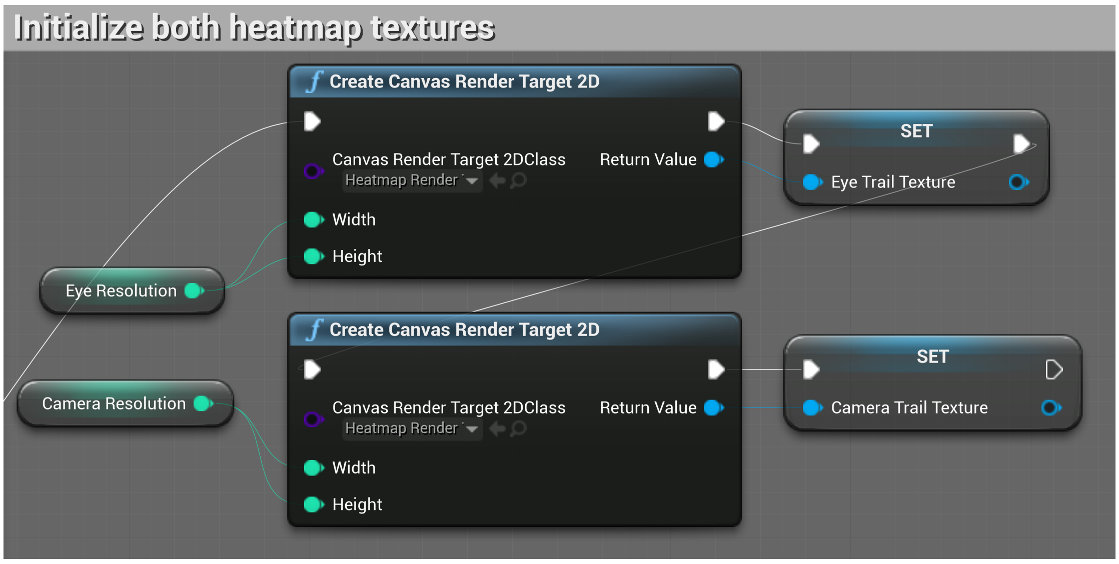

Initialize both heatmap textures
Create Canvas Render Target 2D
Canvas Render Target 2DClass
Heatmap Render
Return Value
Width
Height
Eye Resolution
SET
Eye Trail Texture
Create Canvas Render Target 2D
Canvas Render Target 2DClass
Heatmap Render
Return Value
Width
Height
Camera Resolution
SET
Camera Trail Texture

Figure 5.7 EyeTrackedStaticMeshComponent render target initialisation.

Texture file operations

Another key function of this component is the export and import of heatmap textures. The format is RGBA with 16-bit colour depth for higher accuracy of the measured heatmaps. For this type of texture, the EXR format was chosen. Unreal Engine offers disk input and output operations in this format. The reason for choosing it was that EXR-format images can be exported and imported using the OpenEXR library for Python [65], which will be useful for later image processing.

The whole export depends only on creating a file path and calling the *ExportToDisk* Blueprint function, which takes the export options as a pin. There, it is possible to choose the texture export format and the compression rate; Figure 5.9. The path to the file is created by simple string operations shown in Figure 5.10. The comment of these nodes also contains the format of the path that the operations create. In total, two paths are created. One for the gaze heatmap and one for the camera heatmap.

The loading depends first on loading the image file itself by calling the *Import File as Texture2D* function and passing the filename parameter. The procedure for passing this texture object to an editable render target can be seen in Figure 5.8. A new dynamic material instance of LoadMaterial is created. This material is given a texture parameter with the value of the loaded texture object to be rendered by the *Draw Material To Render Target* function.

EyeTrackingHelperLibrary

This is a static library with Load, Export, and Toggle functions that take care of iterating through all EyeTrackedStaticMeshComponents in the scene and then calling their respective functions individually.

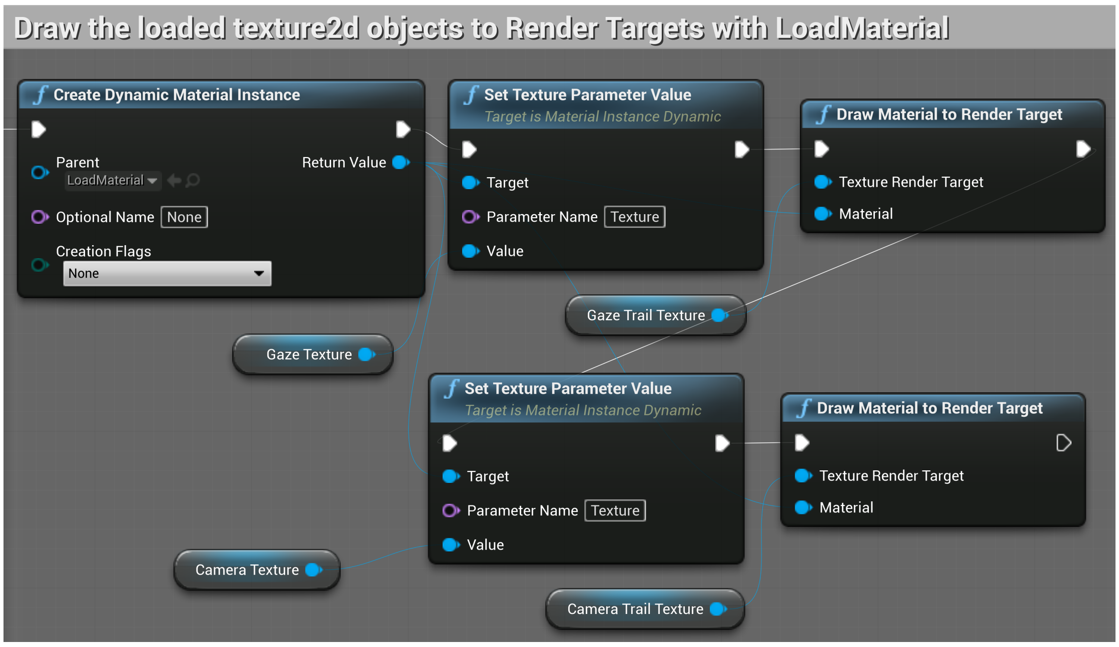

Draw the loaded texture2d objects to Render Targets with LoadMaterial
Create Dynamic Material Instance
Parent
LoadMaterial
Return Value
Optional Name None
Creation Flags
None
Set Texture Parameter Value
Target is Material Instance Dynamic
Target
Parameter Name Texture
Value
Draw Material to Render Target
Texture Render Target
Material
Gaze Trail Texture
Gaze Texture
Set Texture Parameter Value
Target is Material Instance Dynamic
Target
Parameter Name Texture
Value
Draw Material to Render Target
Texture Render Target
Material
Camera Texture
Camera Trail Texture

Figure 5.8 Loading heatmap textures to render targets.

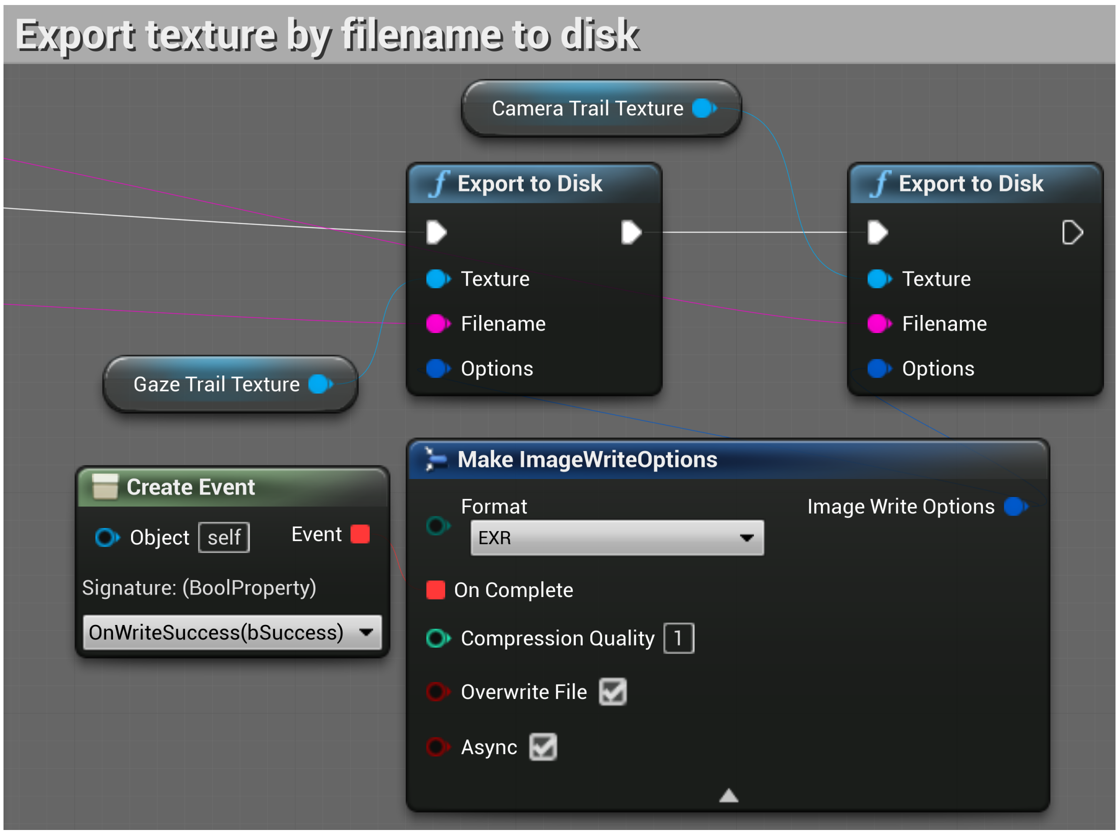


Figure 5.9 Export gaze and camera heatmap textures as EXR image format.

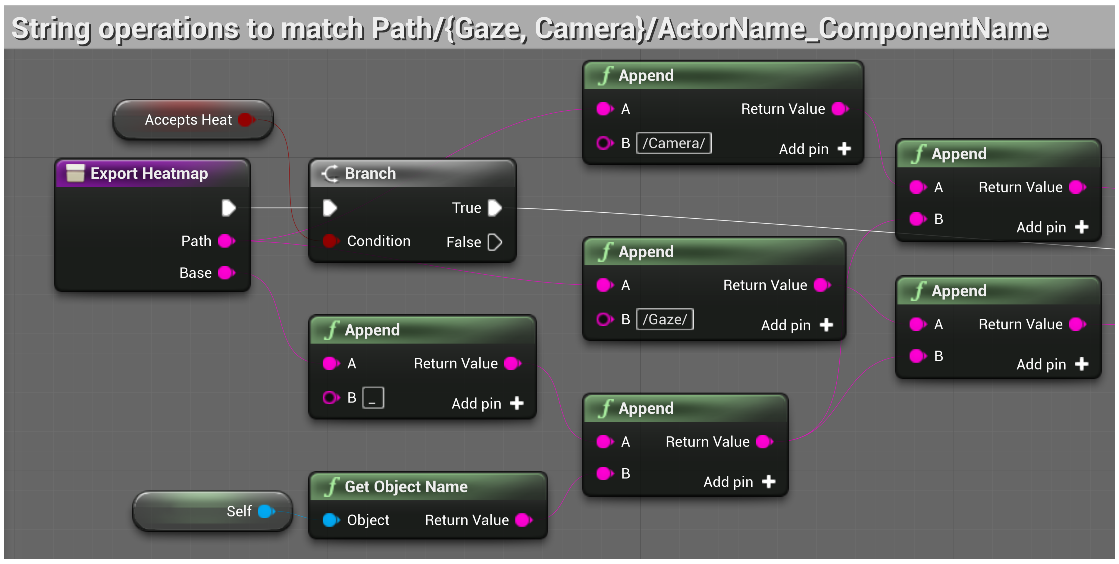


Figure 5.10 Make filepath for the exported gaze and camera heatmap textures.

5.2.2 HeatmapProducer

This class is used to produce heatmaps. Its design is based on the analysis in Section 3.4.7. The blueprint for recording unwrapped meshes is inspired by Tran's tutorial [63]. In order to function, it is absolutely necessary to insert this Actor into the scene.

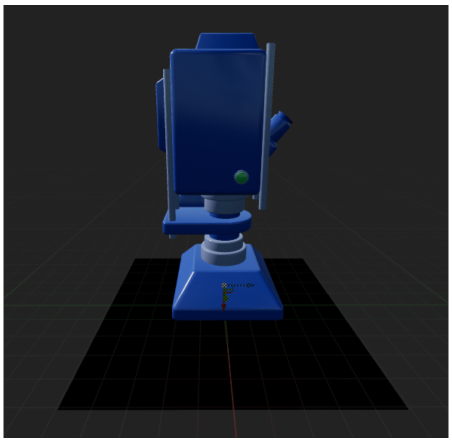
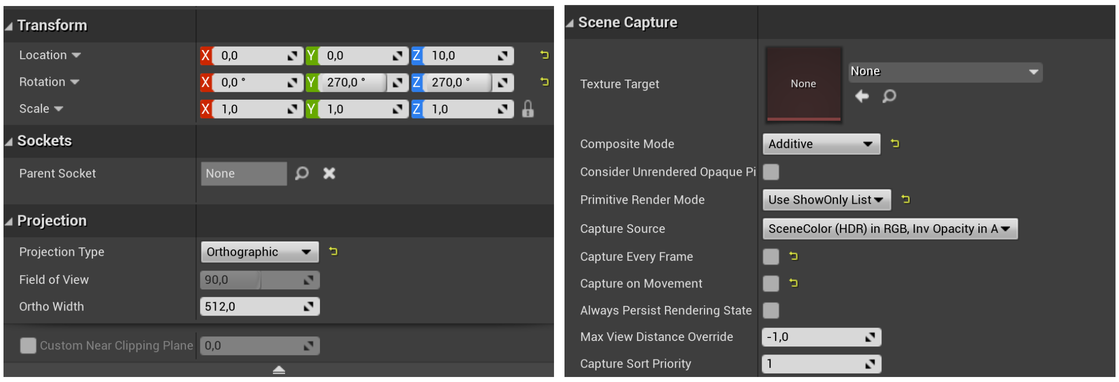


Figure 5.11 HeatmapProducer capture component with background plane in viewport.

First, it is important to describe what the Actor consists of. To record a static mesh that unwraps itself into the world according to its UV map, it is necessary to record the state of this model with some kind of camera. This is what the Scene Capture Component, with the name of *UnwrapUVCapture*, is for. To avoid recording unwanted effects from ambient lights and occlusion, the camera is paired with a simple plane to which a UVRenderBackground material is added, which creates a black background for the mesh, as shown in Figure 5.11. The camera is rotated 270° on the z and y axes to point directly at the background. It is positioned higher than the background, which is shifted to negative values of the z-axis to avoid z-fighting with an unwrapped mesh.

The capture component settings can be seen in Figure 5.12. The camera must be of orthographic type and its width must correspond exactly to the size of the background plane. It is also critical to set the Composite Mode to Additive, to ensure that capturing the scene does not overwrite previous values but adds them to a texture. The background is a square with a 100 cm long edge.



Transform
Location X 0,0 Y 0,0 Z 10,0
Rotation X 0,0 ° Y 270,0 ° Z 270,0 °
Scale X 1,0 Y 1,0 Z 1,0
Sockets
Parent Socket None
Projection
Projection Type Orthographic
Field of View 90,0
Ortho Width 512,0
Custom Near Clipping Plane 0,0

Scene Capture
Texture Target None
Composite Mode Additive
Consider Unrendered Opaque Pi
Primitive Render Mode Use ShowOnly List
Capture Source SceneColor (HDR) in RGB, Inv Opacity in A
Capture Every Frame
Capture on Movement
Always Persist Rendering State
Max View Distance Override -1,0
Capture Sort Priority 1

Figure 5.12 UnwrapUVCapture Component settings.

The HeatProducer Actor contains a private variable of the dynamic material instance of UnwrapBrush material that is initialised in the construction script of this Actor. Its parameters are described in Sections 3.4.7 and 5.1.2.

AlignRenderer function

This function takes as input the texture resolution and the world position of an unwrapped mesh. It is used to align the two components of the Actor and UnwrapBrush dynamic material instance parameters based on texture size and world position; that is, to avoid object culling, Section 3.4.7. To capture a mesh into a texture correctly, the ortho width of the capture component must first be set to the exact resolution of the texture. The background must have its scale values on the x and y axes adjusted to $t/100$, where t is the texture resolution in pixels, to be exactly the same size as the ortho width.

The texture resolution is uploaded to the dynamic material via the *CaptureSize* parameter and the 3D location of the unwrapped object via the *UnwrapLocation* parameter.

PrepareScene

The PrepareScene function just takes an EyeTrackedStaticMeshComponent and includes it along with the black background in the empty show-only list of the capture component.

ApplyHeatToComponent

This function takes care of preparing the HeatProducer Actor and the component for rendering. As Figure 5.13 shows, the black background is first made visible, since it is normally not, then AlignRenderer is called according to the resolution, the PrepareScene function immediately after, and finally the RenderSplatOntoComponent is called to render the heatmap. It finishes by making the background invisible again.

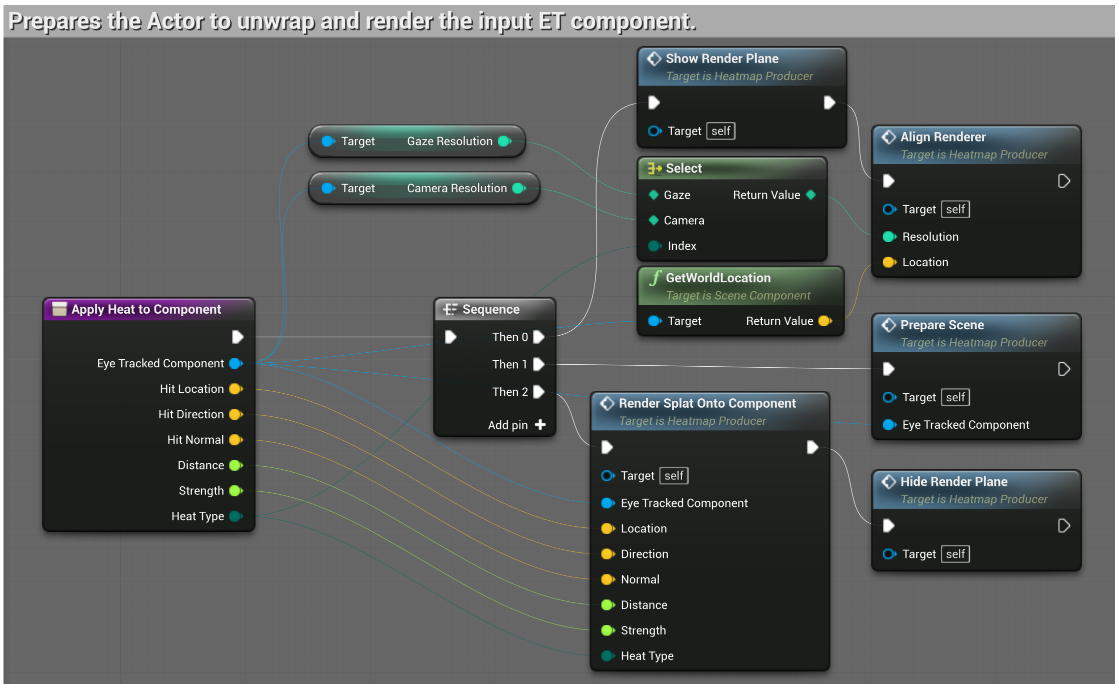
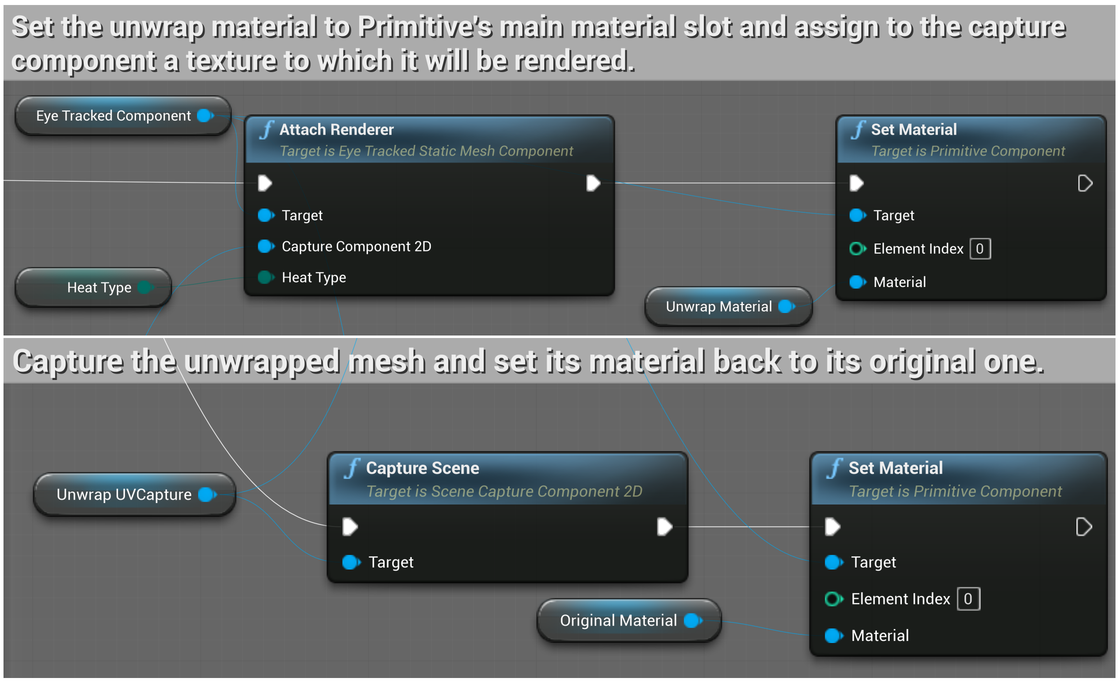


Figure 5.13 HeatmapProducer ApplyHeatToComponent function.

RenderSplatOntoComponent

This is a function of many inputs. Namely, Location, Direction (from camera), Normal, Distance, and Strength. All of them correspond to all the remaining parameters of the UnwrapBrush material, which are set exactly as the previous cases of dynamic material parameters, for example, Figure 5.8. These are HitLocation, DirectionToCamera, HitNormal, BrushRadius, and Strength.

Next, the original material of the component is stored in a local variable, because its material is immediately replaced by the UnwrapBrush instance. This can be seen in Figure 5.14. Thus, it is immediately rendered, after a render target has been assigned to it, using the Capture Scene function. At this point, a new value is added to the render target according to UnwrapBrush. The function ends by returning the component its original material.


Set the unwrap material to Primitive's main material slot and assign to the capture component a texture to which it will be rendered.

Capture the unwrapped mesh and set its material back to its original one.

Figure 5.14 HeatProducer RenderSplatOntoComponent function.

5.2.3 GazeEmitter

It is a child of the SceneComponent class and is crucial for the whole plug-in functionality. It is intended to be added to some user-defined Pawn. After that, the user can define what they want to use as input to this plug-in. If they wanted to track a character moving around the floor of the scene, all they would need to supply GazeEmitter with is the position of the character and a directional vector to the ground. This would cause heatmaps to be created on the floor object, which would be defined as an EyeTrackedStaticMeshComponent.

At the beginning of the game, the component is constructed and in its *BeginPlay* event, it contains the spawn node of the HeatmapProdurer Actor and its assignment to a private variable, as can be seen in Figure 5.15.

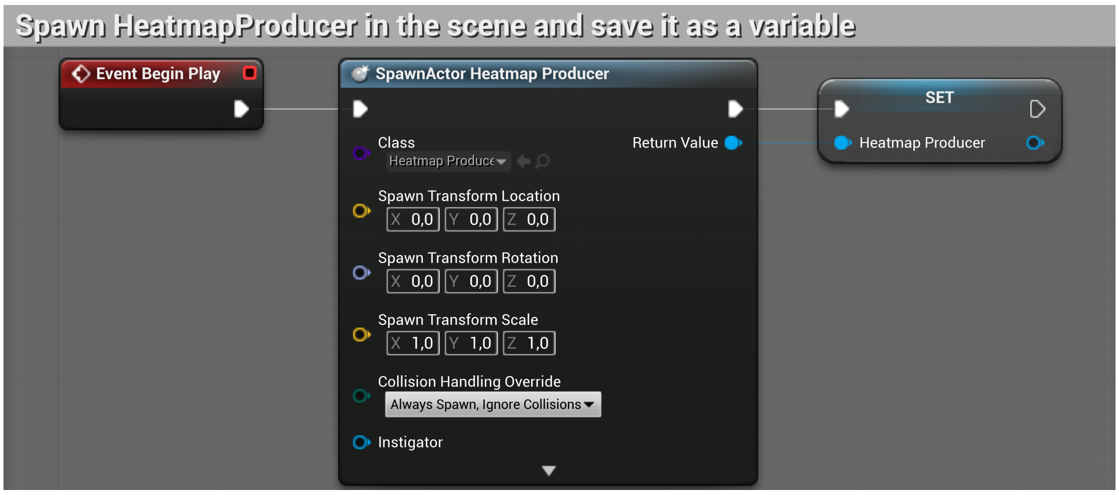


Figure 5.15 GazeEmitter Event BeginPlay.

The whole functionality consists only of calling *LineTraceForObjects*, which passes the start and end of a trace ray that detects a collision on some object in the scene. This is done using the *GetWorldHitFrom* function in Figure 5.16. If it hits something, it first determines if it is an EyeTrackedStaticMeshComponent; if so, the trace was successful.

This first trace was only performed as part of a simple collision to first know which component is being hit. It cannot be guaranteed that the simple collision volume will be detailed enough to represent the entire mesh. If heatmaps were drawn only under these circumstances, it might happen that heat would not be applied to some parts of several objects. Illustrated in the video with the path `video/03-simple-collision-only.mp4`.

There is a function in Unreal called *LineTraceComponent* that takes a component and the start and end of a trace ray as input. Its advantage is that it only performs the tracing on that particular component. Tracing can be set to complex, which will traverse all triangles of a given mesh to find a collision. For meshes with a very high triangle count, this can be a time consuming operation.

This is handled by GazeEmitter's *GetComplexFromSimpleHit* function also in Figure 5.16, which takes a hit as input, extracts the trace ray, and sends it again. The general functionality of creating heatmaps using these functions can be seen in the `video/04-creating-heatmaps.mp4` using a forward vector of a virtual camera.

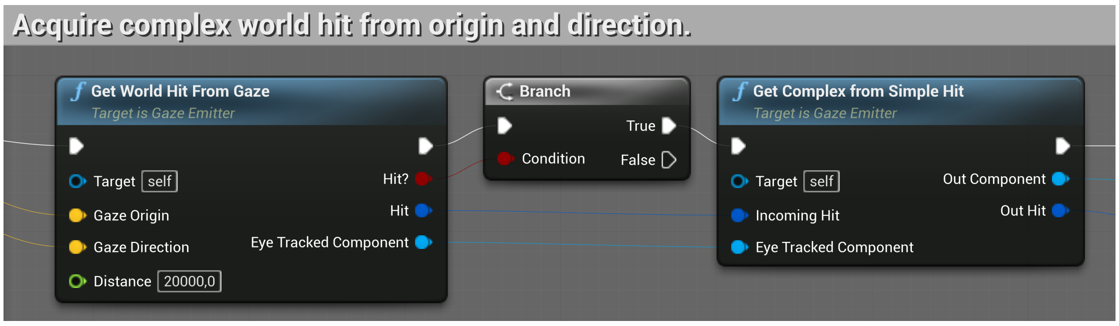


Figure 5.16 GazeEmitter get a complex world hit on a single component.

5.3 Python

The Python section will describe the script for producing the resultant data from the collected EXR textures from different users.

5.3.1 EXR image merger script

This script is used to edit EXR files by iterating through all possible textures of one measurement and one type. This gives a unique texture for all objects that were part of the Unreal scene.

However, these textures exist in multiple variations, depending on how many measurements were made. As a merging script, it ensures that all these textures are merged into one. This is done simply by dividing all texture values by the number of variants. These weighted textures are then merged by a simple add operation.

For this, the OpenEXR library [65] is used to perform read and write operations on EXR files, and NumPy [66] to perform these image operations.

The OpenXR library always reads the image channels as strings. Now, it depends on how the textures are defined. They can be of 32 and 16 bit floats. In the case of exported textures, they are 16-bit EXR images. Using NumPy, this string is converted just according to `numpy.float16`.

The script gets the paths to all texture files of a single scene object. Individually reads their red channel as a string, which is converted to a numpy array. Creates an empty 1D 16-bit float buffer of the size of the texture's pixel count. Then, it successively loads each red channel of all the textures into this buffer, multiplied by a weight constant. The resultant numpy array is converted back to a string and exported by the OpenEXR function, where the red channel is assigned for both the green and blue channels. This can be done because the heatmap trail is a white image. The commented source code can be found in the enclosed media in `src/python/merger.py`.

Chapter 6

Experiment

This chapter will describe the production of the experiment scene. It deals with the transfer of the original model to the Unreal Engine project and finally with the deployment of the plug-in produced in Chapter 5.

6.1 Scene

The scene was made in Blender [67] and transferred to an Unreal project, which can be found in the enclosed media in the `experiment/project/CNB_NC_ET.zip` file. Opening this unreal project may require compilation. All later introduced Blueprints are found in this project. The entire level in the Unreal project can be seen in Figure 6.1.

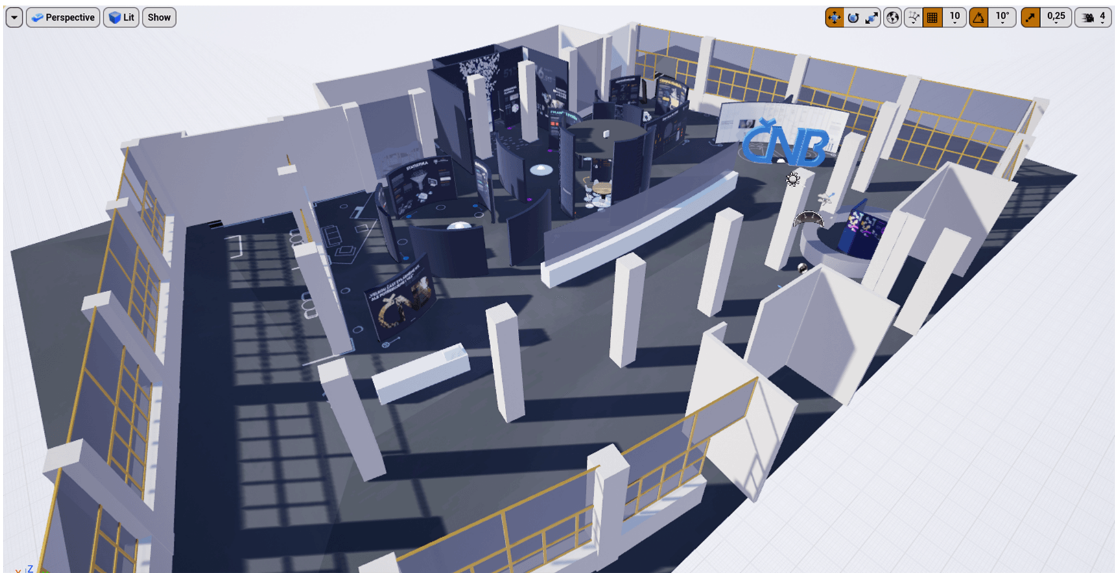


Figure 6.1 NC_ET Level of CNB_NC_ET Unreal project.

6.1.1 Czech National Bank Visitor Centre

The scene produced for the ET experiment is a Visitor Centre exhibition at the Czech National Bank [68]. AV MEDIA SYSTEMS, a.s., which was part of this project, internally created a model of this exhibition in Blender for prototyping purposes [69], the model can be seen in Figure 6.2. This prototype was not involved as part of the construction process. See Figure 6.3 where the exhibition can be seen in a construction process. Ahrend, a.s. was responsible for the realisation of this exhibition in cooperation with AV MEDIA SYSTEMS, a.s..

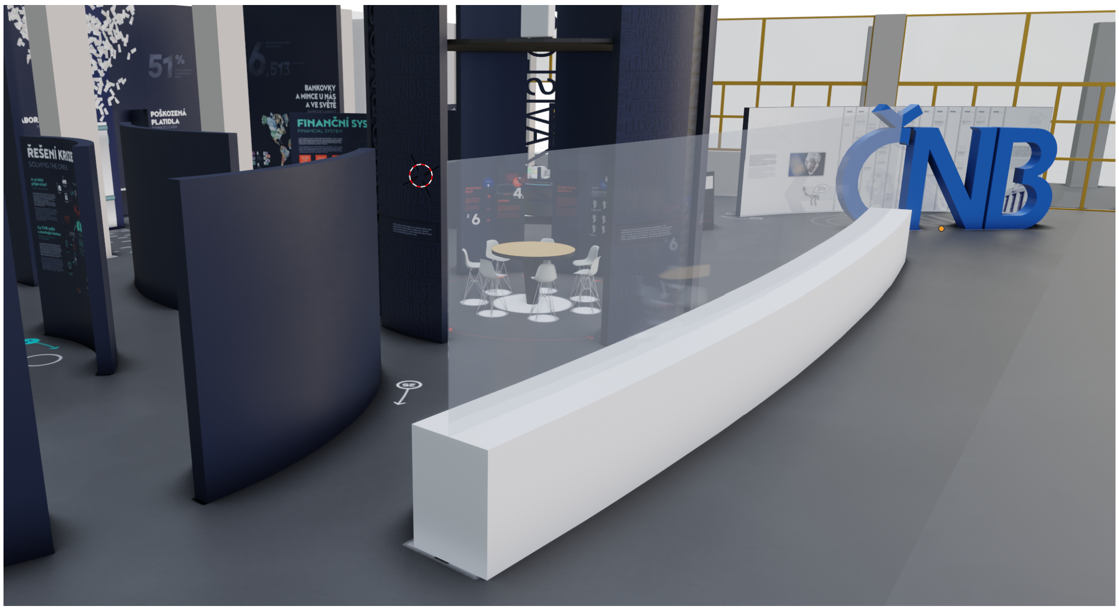


Figure 6.2 AV MEDIA SYSTEMS, a.s. internal prototype Blender model of CNB Visitor Centre. [69]

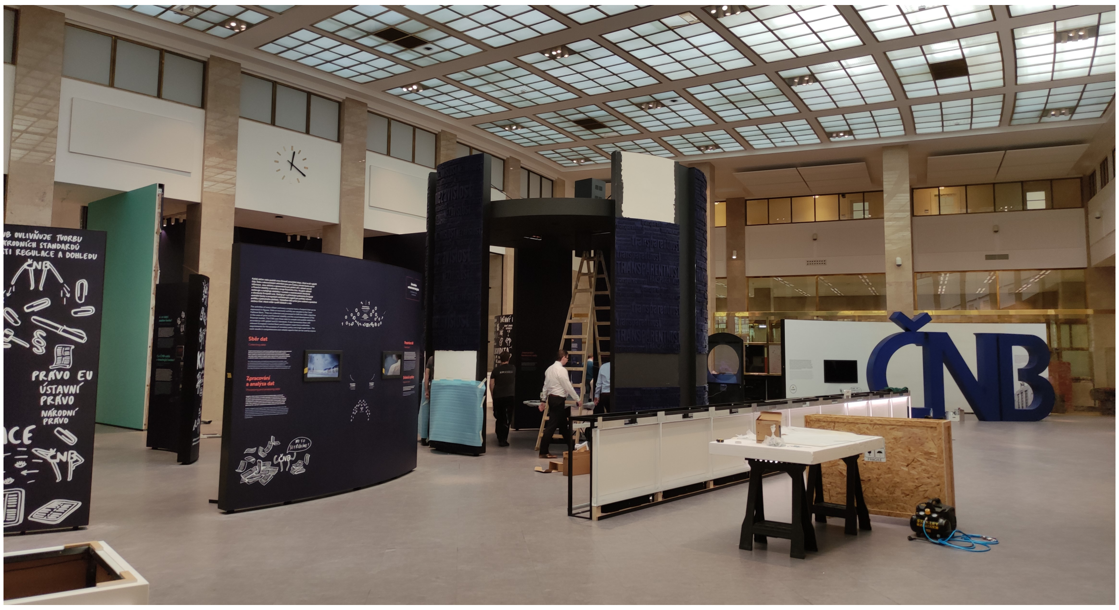


Figure 6.3 Visitor Centre of Czech National Bank during construction.

6.2 Scene modification

The way the Unreal Engine requires the definition of objects in levels along with the definition of the functionality of the created plug-in, it would be impractical to convert this model to the Unreal scene in its original state. This section defines issues with the Blender scene, which will be modified to allow the scene to be converted into an Unreal project [70]. The very first aspect that had to be modified was to separate large models with disconnected meshes into multiple models with their respective material. The walls and information panels are joined, as illustrated in Figure 6.4. And the second problem with most objects in the scene, as can be seen in Figure 6.5, was that they had defined overlapping UV maps that contained more than one material.

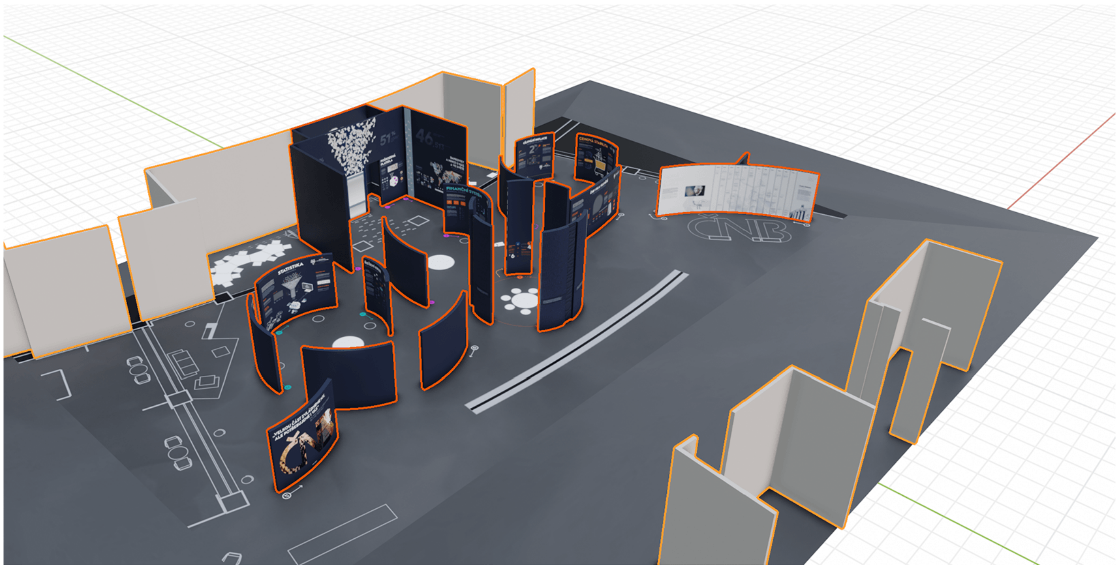


Figure 6.4 Multiple meshes joined into one model.

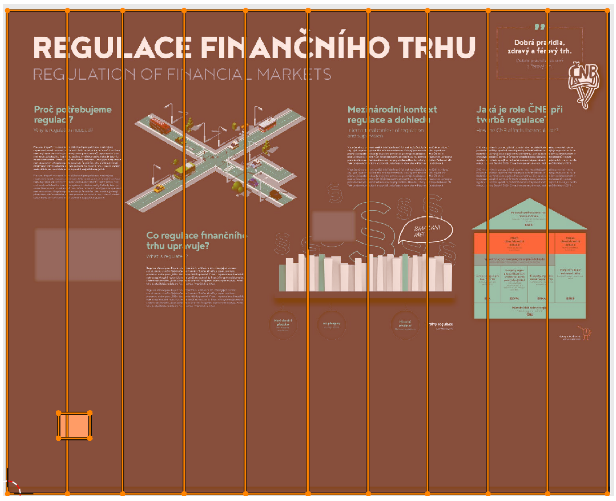

REGULACE FINANČNÍHO TRHU
REGULATION OF FINANCIAL MARKETS
Proč potřebujeme regulaci?
Mezinárodní kontext regulace a dohledu
Jaká je role ČNB při tvorbě regulace
Co regulace finančního trhu upravuje?
Dobrá pravidla, zdravý a férový trh.

Figure 6.5 An overlapping UV map on one information panel.

6.2.1 Joined objects

The edit mode had to select all surfaces to be separated from the shared model and then perform the operation shown in Figure 6.6a, separate by selection. This object then became independent in the scene. Next, after this separation, it was worth using the function again, but this time by material to remove loose material definitions left over from the other meshes. This was done for all whole meshes in the model.

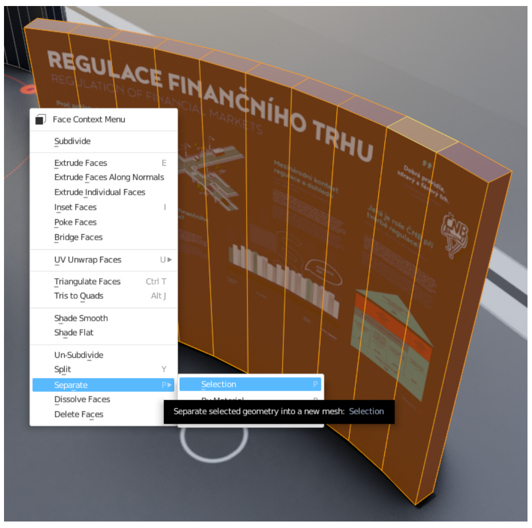
Each information panel was defined by two materials. There was always a material that displayed the information on a panel, but there was also a material that used colour on its back. This colour was defined in panel 2. All the backs of all the panels had to be redefined to their own material to match the original colour from panel 2.

6.2.2 Overlapping UVs

When all meshes are separate, it is important to ensure that heatmaps can be created on these objects according to the algorithm in Section 3.4.7. It is necessary that these objects have non-overlapping UV maps defined to guarantee uniqueness for all mesh faces.

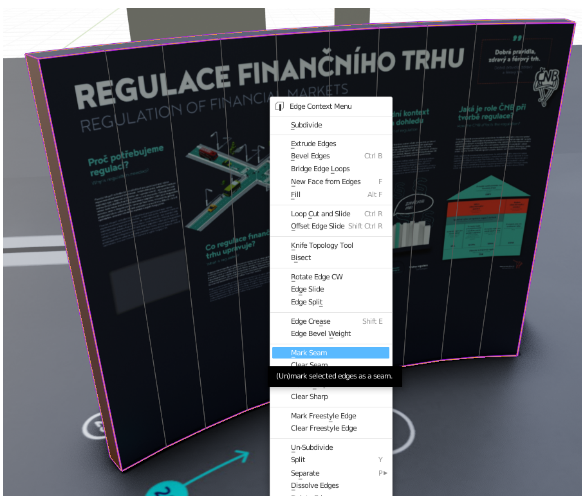
This is done in the Object Data Properties setting of the object. A new map is added in the UV Maps section and it is important to keep the original one for the next step. This new UV map can be defined using a Blender feature in edit mode called Smart UV Project, which will generate the UV map itself according to the specified parameters. However, it is not recommended to do this without first manually defining seam edges, as, for some objects, this automatic generation could potentially yield poor results. Marking an edge as a seam can be seen in Figure 6.6b.

Some objects capable of being in the Unreal project with the plug-in have no UV maps defined or use a basic material. These are simpler cases for which a definition of a new UV map is sufficient.



Separate selected geometry into a new mesh: Selection

(a) Separate selection of faces into new mesh.



(Un)mark selected edges as a seam.

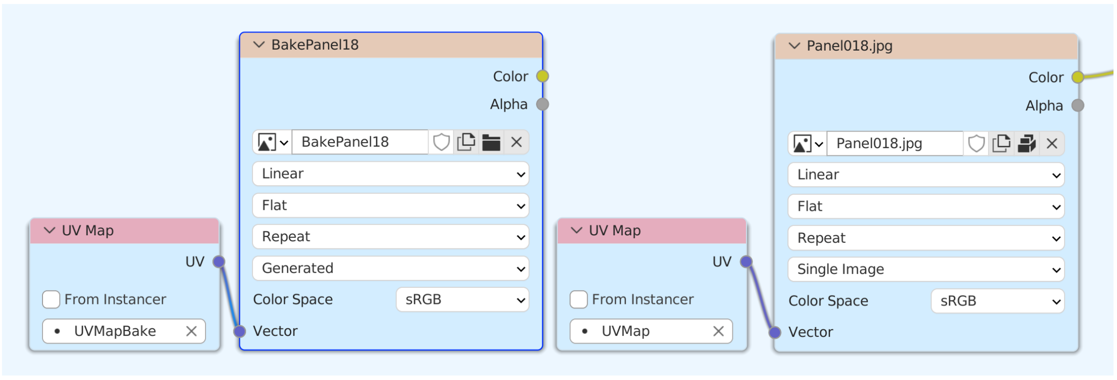
(b) Mark mesh edges as seam for UV unwrap.

Figure 6.6 Blender edit mode separate selection and mark seam operations.

6.2.3 Material Baking

This step describes the Blender procedure to convert a material defined for one UV map to another using texture baking. Also described in this tutorial [71].

The method consists of defining a new UV map on the same object. The complete setup can be seen in Figure 6.7. Then, in the Shader Editor, add a new *ImageTexture* material node through which a new texture is created without an alpha channel of square resolution. This node is connected to the material node *UV Map*, which links to the newly created *UVMapBake*. To be absolutely sure, the original UV map can be linked to the original texture.


UV Map
UV
From Instancer
UVMapBake

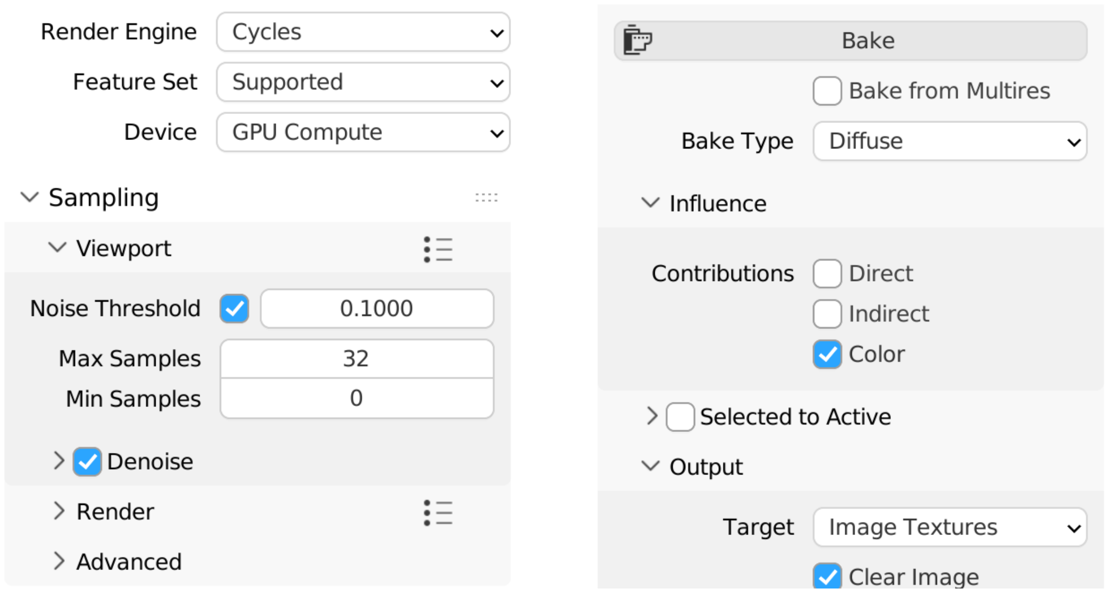
BakePanel18
Color
Alpha
BakePanel18
Linear
Flat
Repeat
Generated
Color Space sRGB
Vector

UV Map
UV
From Instancer
UVMap

Panel018.jpg
Color
Alpha
Panel018.jpg
Linear
Flat
Repeat
Single Image
Color Space sRGB
Vector

Figure 6.7 Preparation of the information panel material nodes for baking into a new texture.

To bake into UVMapBake, it is necessary to open the Bake settings of Blender's Cycles renderer and select the corresponding ImageTexture node. Cycles settings are in Figure 6.8. It is enough to set the diffuse *BakeType* that will not include direct or indirect light from the scene, only colour. The resulting texture can be saved or embedded in the project. The difference between the baked texture and the original texture can be seen in the Appendix C.


Render Engine Cycles
Feature Set Supported
Device GPU Compute
Sampling
Viewport
Noise Threshold ✓ 0.1000
Max Samples 32
Min Samples 0
✓ Denoise
Render
Advanced

Bake
☐ Bake from Multires
Bake Type Diffuse
Influence
Contributions ☐ Direct
☐ Indirect
✓ Color
☐ Selected to Active
Output
Target Image Textures
✓ Clear Image

Figure 6.8 Blender cycles settings for baking simple diffuse colour onto an ImageTexture.

6.2.4 Origin of models

The original model contains many grouped objects that share a common origin or have their origin right at the origin of the scene. This needs to be changed for those objects that will be exported, due to Unreal Engine's rendering pipeline, which culls in one step those objects that are not visible in the view frustum. This happens during heatmap rendering when the object is somewhere else than where the Scene Capture Component is looking. Section 5.2.2 describes how the render Actor changes its position in the world based on the position of the object being rendered. Because of this, the mesh itself must be guaranteed to be near its origin for at least the distance determined by the size of the rendered heatmap texture, which also determines the width of the orthographic frustum.

In Blender, it is worth setting the length of the units to centimetres in Scene Properties first to mimic the unit system in Unreal. Origin can be changed in the context of this scene in the following style; generate a new origin at the centre of the object geometry, place a 3D cursor on this new origin and set the z-coordinate to 0 so the cursor will sit on the ground, and finally assign the object's origin to this 3D cursor.

6.2.5 Model export

An object with a newly defined UV map, a material that contains the newly baked texture of the original material, can be exported to Unreal Engine. It includes tools that allow some 3D modelling programmes to export their entire scenes at once. This is called *Datasmith*. This was only available in the past using an unofficial Blender plug-in, which was not supported and could not be used at the time of writing the thesis. This route cannot be taken; the conversion of the whole scene is done manually.

Exporting an object from Blender as an `.fbx` file allows one to bundle its mesh, material, and texture with it. Procedural materials cannot be exported in this manner and must be baked before.

6.3 Unreal project

This section will describe how the experiment scene was constructed by adding exported models one by one from Blender. The entire project can be found in the enclosed media in.

6.3.1 Setting up the scene

The models were inserted into the scene inside the logical entities represented by different Actors in the project directory in the Map folder, Figure 6.9.

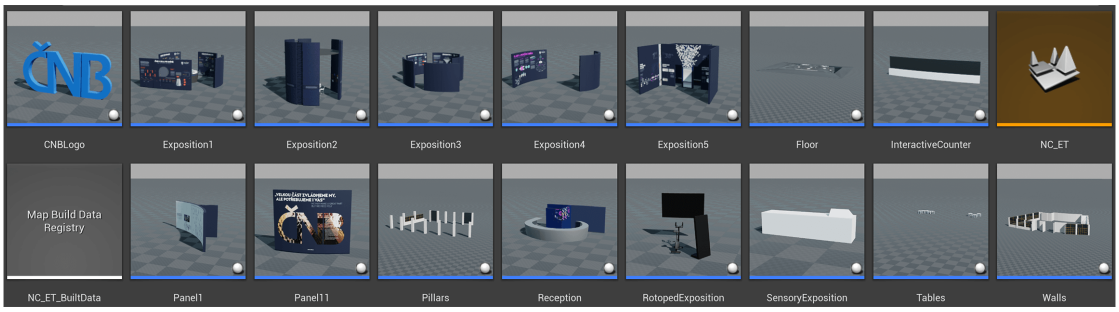


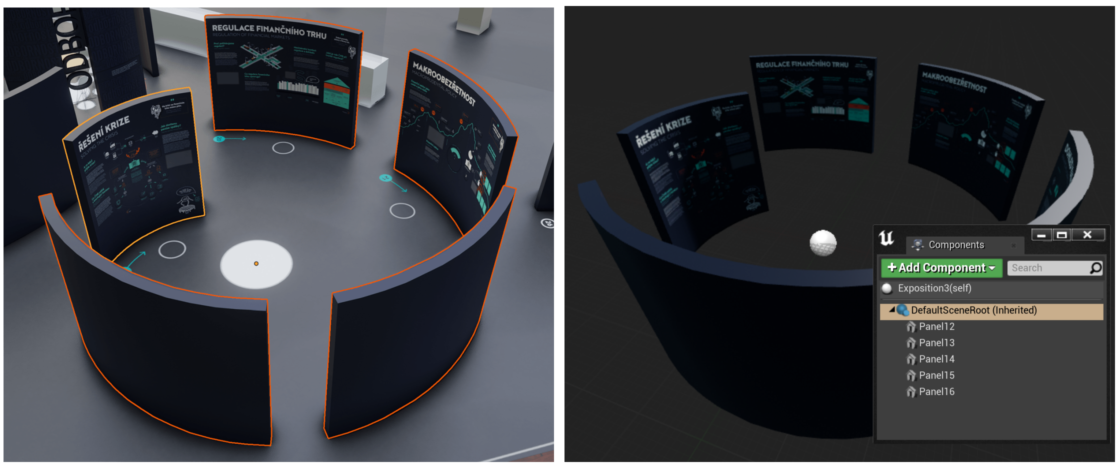
Figure 6.9 Actors containing scene objects.

Asset placement

An Fbx file can always be imported into Unreal with a simple drag-and-drop. At most, such an object brought StaticMesh, Material and Texture2D with it into the project. These Assets are, respectively, divided into Geometries, Materials, and Textures directories. All imported models are in their local coordinates, guaranteed by setting up their origins according to Section 6.2.4.

For a logical scene entity such as an exhibit, a single Actor was used. For this actor, several components of the EyeTrackedStaticMeshComponent class defined in Section 5.2.1 were added. Those were manually placed in the local world coordinate system of a given Actor exactly according to the coordinates that the meshes would have in Blender relative to the origin of an exhibit. See Figure 6.10. Each of these components has been assigned a StaticMesh. Next, the texture resolution and the *AcceptsHeat* boolean are set.

Not all objects from the scene in Blender were used to keep the experiment relatively clean. The reason is the recording of both heatmaps from the eye tracker and the virtual camera. The simpler one has a much larger brush radius set, so it is worth demonstrating it on objects that are large.



(a) Panels with origin at exhibit centre in Blender. **(b)** Panels defined around Actor's origin in UE.

Figure 6.10 Exhibit defined as logical scene entity.

Collisions

Each StaticMesh can be collided with using its complex collision, which is in effect just a collision against all its faces. It is possible to define simple collisions for the mesh in the Static Mesh Editor in Unreal Engine to reduce the complexity of collision calculations. For all static meshes imported from Blender, the Convex Decomposition function in the editor was used to generate a simple collision volume.

6.3.2 Pawns

In the scene, two Pawns are defined to move around the scene. One is used to collect data, while the other is used only to view them.

VRCollector

This pawn is a pure pawn without any movement ability. It contains the GazeEmitter component described in Section 5.2.3 and a camera that is directly locked to an absolute HMD position obtained from a positioning system. The problem with these coordinates is that they do not move with a pawn in the world; they move with it. To keep them positioned in the same way as the camera, it is necessary to fix the pawn at the origin of the world from which it will not move. To solve this problem, the pawn will not traverse the scene, but the entire scene will move around the pawn. Specifically, in the opposite direction to the way the pawn wants to move. See Figure 6.11. In the whole scene, an Actor *SceneOrigin* has been created who is superior to all other Actors.

Its main functionality, however, is that it starts an experiment that lasts three minutes. When finished, all textures are saved in the folder `C:/ProgramData/NC_ET/` and the programme is exited, as can be seen in Figure 6.12. During this experiment, data were collected using the GazeEmitter component, Section 5.2.3. Figure 6.13 shows the Tick Event of this Pawn, which always asks if the experiment is running. If it does, it sends the forward vector of the Camera from its origin as a ray classified as Camera ray type to the *Emit Ray From* function, defined in Section 5.2.3. It can also be seen in the figure that the same vector is sent for the ray classified as Gaze. The reason for this is discussed in Section 6.5. The implementation of this cycle corresponds exactly to the activity diagram in Figure 4.1 in Section 4.3.

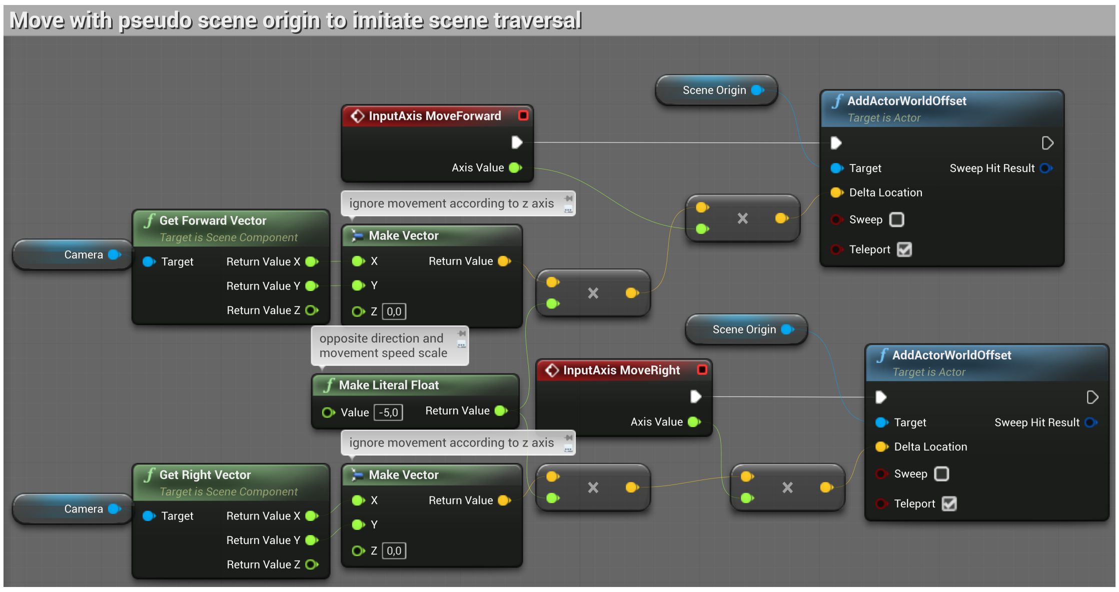


Figure 6.11 VRCollector movement definition.

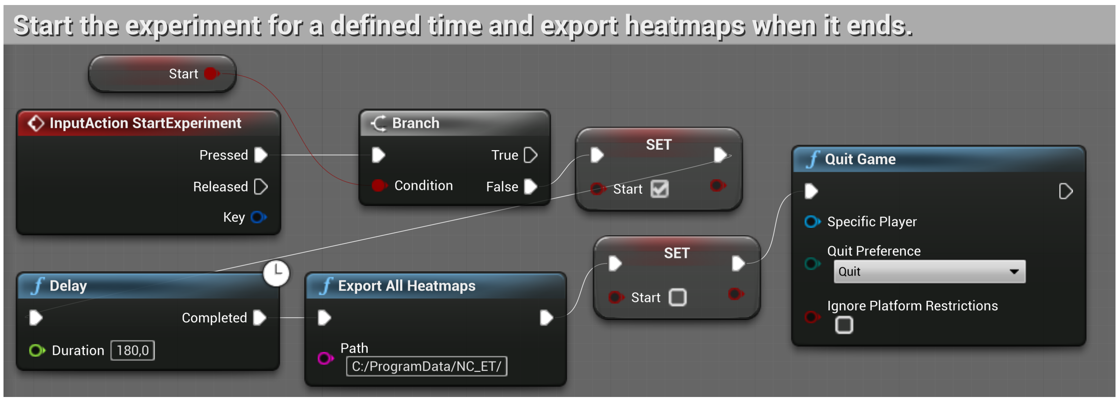


Figure 6.12 VRCollector set experiment start and duration.

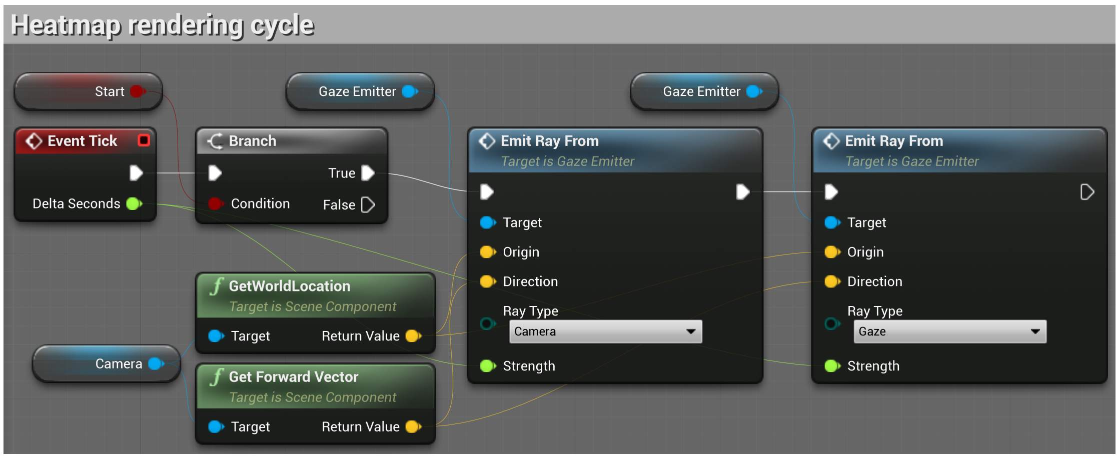
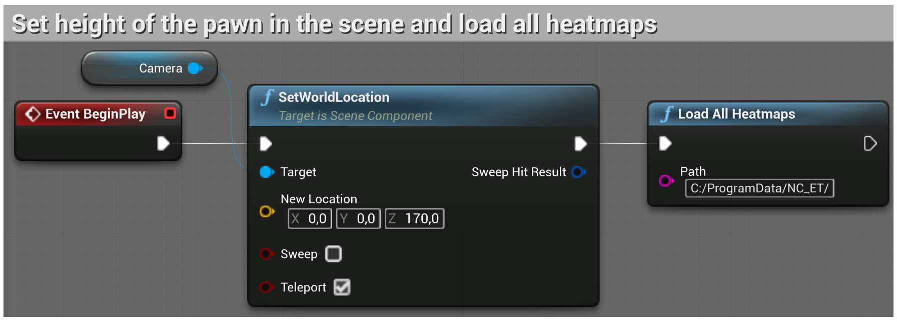


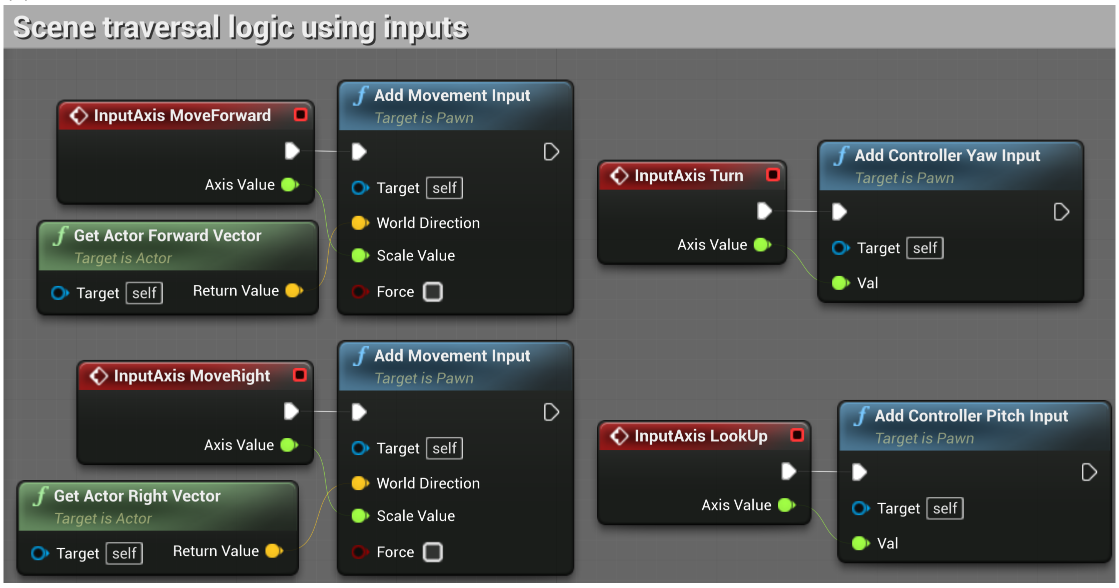
Figure 6.13 VRCollector Tick Event sending rays from virtual camera.

SceneObserver

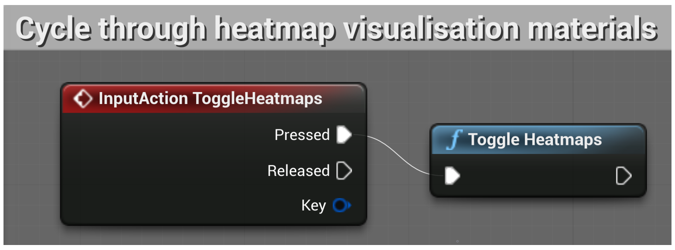
This pawn allows one to view the scene after the heatmaps have been measured. This is a pawn of type Character, so it contains its own collider and basic movement functionality. Unlike the VRCollector, this is a pawn that traverses the scene conventionally and bumps into objects. In Figure 6.14a its initialisation can be seen, where it specifies its height and loads all heatmaps that are stored in the `C:/ProgramData/NC_ET/` directory structure. Use the H key to switch between heatmap visualisations. There are three different options. First, the original material without heatmaps, then the TrailCompare from Section 5.1.4 and finally the HeatMaterial from Section 5.1.3.



(a) Initialise character and load heatmaps.



(b) Traversal character logic.

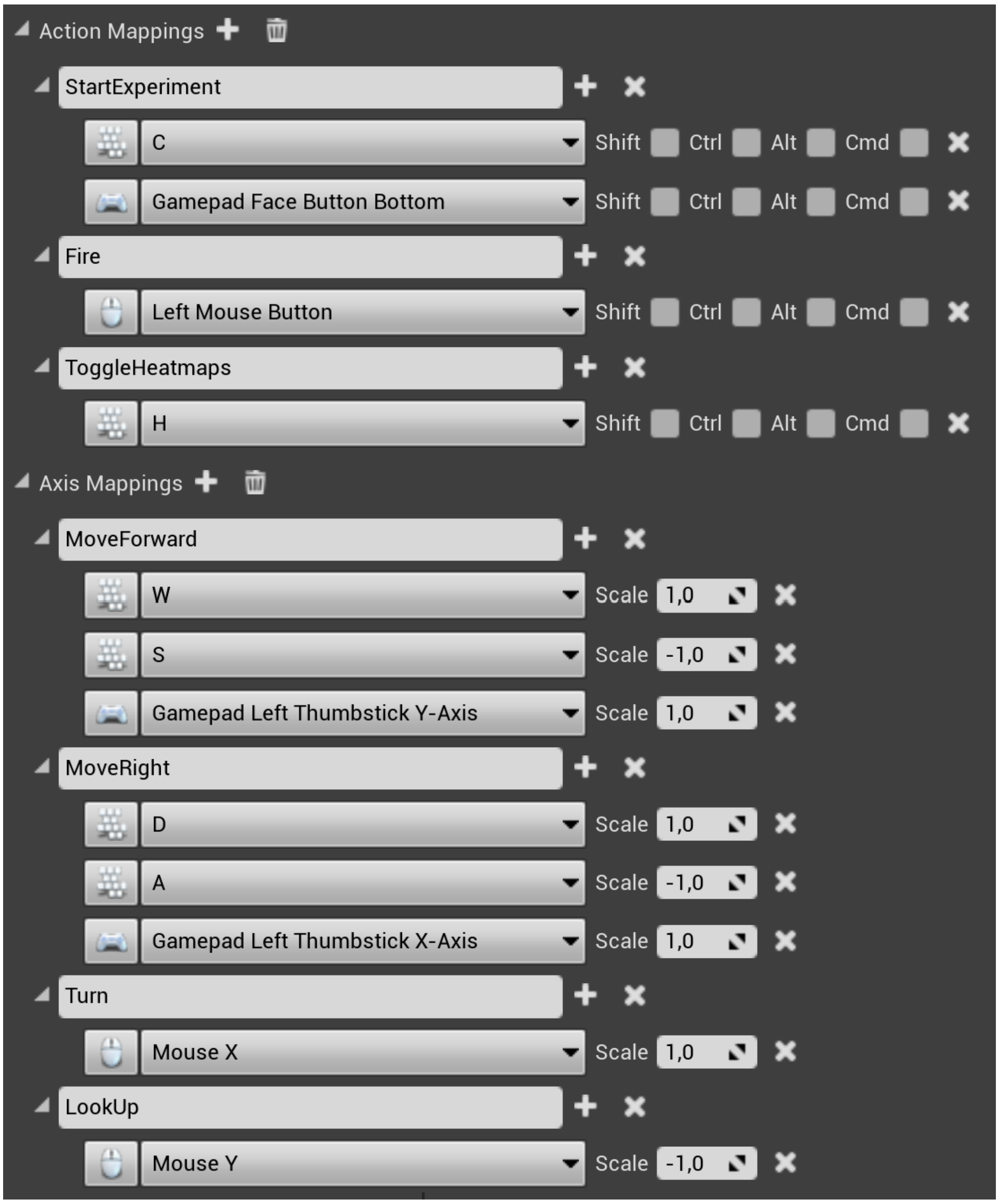

Cycle through heatmap visualisation materials
InputAction ToggleHeatmaps
Pressed
Released
Key
Toggle Heatmaps

(c) Toggle heatmap visualisation.

Figure 6.14 SceneObserever Event Graph.

6.3.3 Controls

Both Pawns use this simple control interface. All button mappings can be seen in Figure 6.15.



Action Mappings
- StartExperiment
  - C — Shift ☐ Ctrl ☐ Alt ☐ Cmd ☐
  - Gamepad Face Button Bottom — Shift ☐ Ctrl ☐ Alt ☐ Cmd ☐
- Fire
  - Left Mouse Button — Shift ☐ Ctrl ☐ Alt ☐ Cmd ☐
- ToggleHeatmaps
  - H — Shift ☐ Ctrl ☐ Alt ☐ Cmd ☐

Axis Mappings
- MoveForward
  - W — Scale 1,0
  - S — Scale -1,0
  - Gamepad Left Thumbstick Y-Axis — Scale 1,0
- MoveRight
  - D — Scale 1,0
  - A — Scale -1,0
  - Gamepad Left Thumbstick X-Axis — Scale 1,0
- Turn
  - Mouse X — Scale 1,0
- LookUp
  - Mouse Y — Scale -1,0

Figure 6.15 Control scheme of the unreal project with experiment.

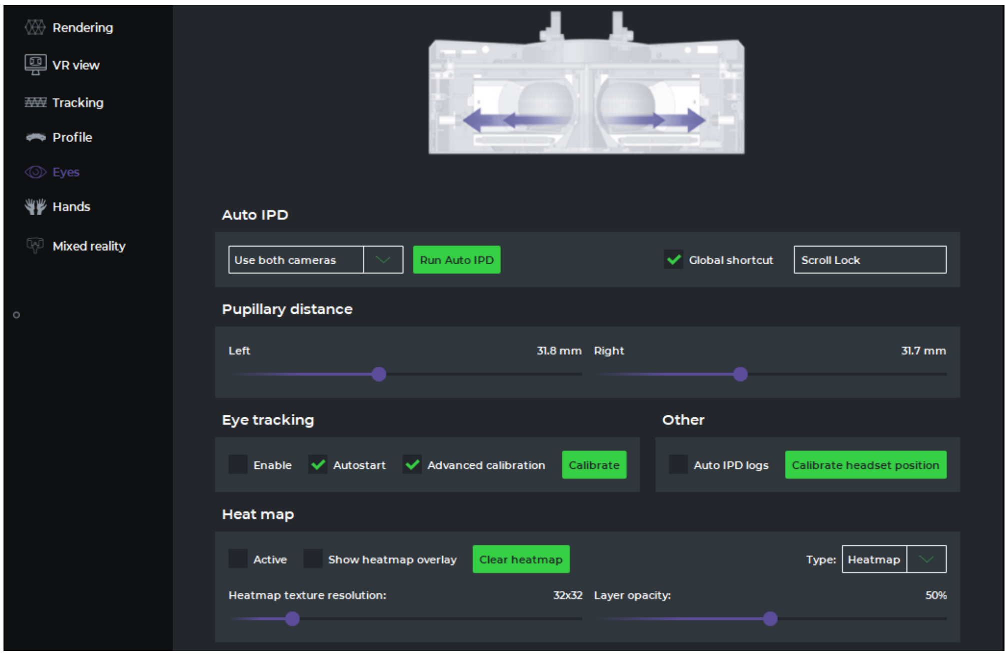
6.4 XTAL

This section describes the setup of the XTAL headset within the 3D CAVE laboratory at the Faculty of Mechanical Engineering of CTU. It also presents issues with the current beta OpenXR runtime by Vrgineers [72] and its unsuccessful troubleshooting. Note that the lab has an older generation of XTAL than the current XTAL 3.

6.4.1 Setup

The laboratory has set up several VICON tracking cameras that collectively acquire the absolute position of defined objects in this space using small reflective spheres. Several of these spheres are attached to XTAL. The VICON system obtains the absolute position of these spheres and sends them as an overall object defined as *Xtal01* over the network. In the settings of the VR Tool runtime, which is the driver for the XTAL headset [73], the positioning system can be set to VICON. There, it can specify the IP address of the system that sends these positioning data.

Regarding XTAL itself, Figure 6.16 shows the eye runtime settings. Functions such as auto-IPD can be triggered, which is measured using ET to automatically adjust the lens distance inside. There is also a calibration section for the ET. The user must always run a new calibration when a programme is started. Advanced calibration is a seven-point calibration that results in more precise measurements.



Auto IPD

Use both cameras | Run Auto IPD | ✓ Global shortcut | Scroll Lock

Pupillary distance

Left 31.8 mm | Right 31.7 mm

Eye tracking

Enable | ✓ Autostart | ✓ Advanced calibration | Calibrate

Other

Auto IPD logs | Calibrate headset position

Heat map

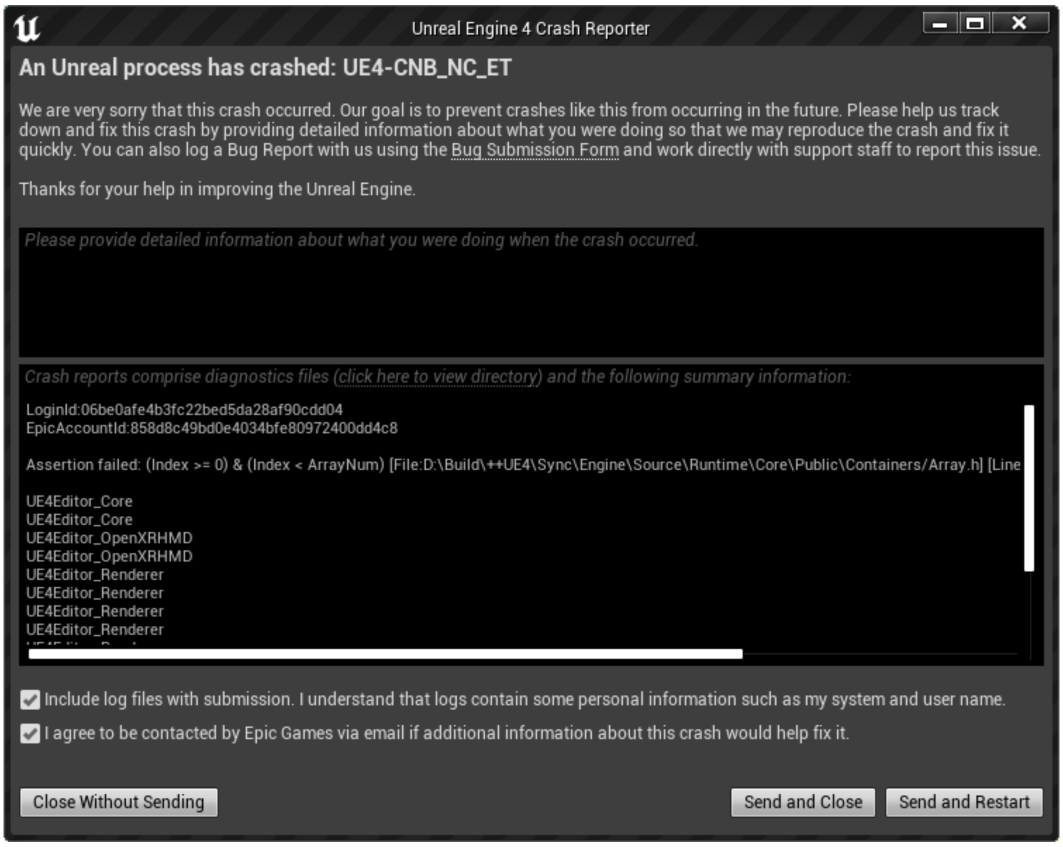
Active | Show heatmap overlay | Clear heatmap | Type: Heatmap

Heatmap texture resolution: 32x32 | Layer opacity: 50%

Figure 6.16 Vrgineers VR Tool runtime eye settings. [73]

6.4.2 Troubleshooting

The main problem that occurred was the instability of the OpenXR XTAL runtime, which caused regular crashes of the entire UEEditor during the VR Preview in the Unreal project. The VR Preview sometimes lasted for a full three minutes before crashing and sometimes did not even turn on. The error message can be seen in Figure 6.17.



An Unreal process has crashed: UE4-CNB_NC_ET

We are very sorry that this crash occurred. Our goal is to prevent crashes like this from occurring in the future. Please help us track down and fix this crash by providing detailed information about what you were doing so that we may reproduce the crash and fix it quickly. You can also log a Bug Report with us using the Bug Submission Form and work directly with support staff to report this issue.

Thanks for your help in improving the Unreal Engine.

Please provide detailed information about what you were doing when the crash occurred.

Crash reports comprise diagnostics files (click here to view directory) and the following summary information:

```
LoginId:06be0afe4b3fc22bed5da28af90cdd04
EpicAccountId:858d8c49bd0e4034bfe80972400dd4c8

Assertion failed: (Index >= 0) & (Index < ArrayNum) [File:D:\Build\++UE4\Sync\Engine\Source\Runtime\Core\Public\Containers/Array.h] [Line

UE4Editor_Core
UE4Editor_Core
UE4Editor_OpenXRHMD
UE4Editor_OpenXRHMD
UE4Editor_Renderer
UE4Editor_Renderer
UE4Editor_Renderer
UE4Editor_Renderer
```

☑ Include log files with submission. I understand that logs contain some personal information such as my system and user name.

☑ I agree to be contacted by Epic Games via email if additional information about this crash would help fix it.

Figure 6.17 UEEditor crash log after OpenXR VR Preview.

The latest version of OpenXR runtime from Vrgineers [72], sent via email, was tested. The runtime was still in beta at the time of writing, and it was not possible to resolve this issue for Unreal Engine 4.27.

Support from Vrgineers was consulted on this issue, and they said that they are also experiencing runtime crashes of the UEEditor for unknown reasons, but not at such frequent intervals [43]. It has been tested to see if the debugging tools used in UEEditor are to blame for this. Although the crashes were not as sudden, the project always lasted at least half a minute; it did not solve the problem.

While the VR Preview was running, it was possible to calibrate the ET and send rays out into the scene. The problem is that this process is rather long, and before the participant could get to the measuring part, the whole programme would crash. It was tested to see if it would crash even when running the standalone version of the game. In this case, it was not determined whether this would solve the problem because the OpenXR runtime does not initialise the XTAL headset to display the scene inside, even though the Start in VR option is enabled in the project settings. The console command "*stereo on*" for the Blueprint node *Execute Console Command* was also tried, but nothing changed. It will run only for the VR Preview in the UEEditor.

The Vrgineers support recommended migrating the entire project to Unreal Engine 5, as it contains a much more stable implementation of OpenXR communication. This was unsuccessful; details can be seen in the Appendix A.

It was decided, after consultation with the thesis supervisor, to use the Vrgineers stable SDK plugin for Unreal Engine [74], which includes the original native runtime. However, this does not allow the use of ET.

6.5 Data collecion

Due to the instability of the OpenXR runtime, the originally intended eye tracking experiment could not be performed, so it was decided that the data would only be collected using a forward vector camera in a virtual scene in which the participant would walk around using the XTAL headset.

The plug-in created in this thesis, Chapter 5, contains raycasting functionality that can be easily used in any Unreal project. Thus, it already depends only on using a working VR headset with ET. It has been tested on an unstable XTAL runtime that the feature works and sends rays into the scene. It was not possible to use any other headset as an alternative due to the unavailability of other headsets and the lack of time to incorporate the changes into the thesis.

The test experiment will be carried out as originally intended, but without the use of ET. It will collect both Gaze and Camera heatmaps, although it will only emit rays from the forward vector of the camera. This will serve as a demonstration of what it would look like if working ET hardware was used.

6.5.1 Test scenario

The test scenario used in this experiment is a scene traversal by a person who has not previously seen it. Each participant creates their own collection of heatmaps. These heatmaps will be used to determine which object most interested the group of participants or where the most textures intersected their trails.

6.5.2 Measurement process

One measurement consisted of first placing a headset on the participant to set their PID. Then, the Unreal Engine project was launched with the VRCollector pawn, defined in Section 6.3.2. The XTAL in the lab is connected to the computer, which is used as a backpack, to walk around the lab. The computational part of the backpack had to be unplugged from the charging dock and attached to its other part. The backpack was put on by the participant with the headset. Before this, the participant was instructed how to move around the scene using a game controller. They could have moved around without a controller, but they would soon have hit a wall in the lab because the exhibition is much bigger, so a controller was necessary. The speed of movement resembled a fast walk. They were also informed that the scene was an exhibit of the Czech National Bank. After ensuring that the participant was ready, a test period of three minutes was started. During this time, the participant was free to walk around the scene as they wished. A participant was looking at the scene that was with its original materials. Heatmaps were produced in the background. At the end of the three minutes, all heatmaps were saved in the `C:/ProgramData/NC_ET/` path and the programme was stopped.

Videos of the measurements were recorded. Participants 4 and 8 were chosen to demonstrate the experiment. Their recorded session is in the enclosed media in the `video` folder. All heatmaps from all participants are compressed in the `experiment/data/ParticipantData.zip` file.

6.6 Resultant data

A total of 12 measurements were made, so 12 different heatmap collections were produced. These data were grouped into the DATA folder and each measurement was in the folder of a numbered participant. Each participant contains a Camera and Gaze folder, due to the two different types of heatmap. The Python merger script from Section 5.3 was put in the same folder, which merged these participants into folder `ResultantData`. The same folder is in the enclosed media in the `data` folder.

There are several information panels in the exhibition scene. The resultant data showed that they were the majority target of interest. It is possible to turn on the Unreal Project and view the data by running the scene with the SceneObserver pawn. The data folder structure, along with the Gaze and Camera folders, should be put in the `C:/ProgramData/NC_ET/` folder to view them. The scene traversal with the resultant data can also be viewed in `video/05-resultant-data-showcase.mp4`.

From the resultant data, the four panels that showed the highest trail densities were selected.

Panel11

In Figure 6.18 it is possible to see the difference in the textures collected of the *Panel11* scene object. Panel11 has the highest intensity of its trails of all the objects in the scene, as shown in Figure 6.19b. The density, on the other hand, is not large and it can be seen that large brushes were used to produce it. This means that the participants were looking at the object from a longer distance. Particularly at the beginning of the scene while looking around to the left or right, this is one of the first objects visible in the scene. Five participants did not even see the object. Only three examined it. Participant 7 is an outlier who looked at the object for much longer and contributed greatly to the result.

Panel14

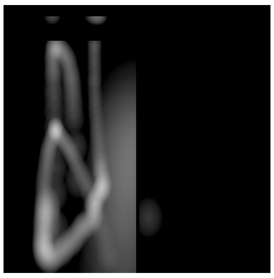
Panel14 in Figure 6.20 shows the different variations of the heatmaps collected from the participants and immediately Figure 6.21 shows the resulting texture and its comparison with the original material. Only three participants have not seen this object. Unlike Panel11, this time the object was not only viewed from a distance, but also seen from up close by five participants. Since it has much higher density of different twisting trails.

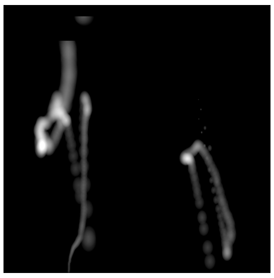
Panel17

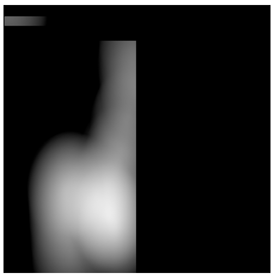
Like the previous cases, the results are illustrated in Figures 6.22 and 6.23. This is the same case as the Panel14. It was examined up close. Compared to the rest, it shows in its trails that more participants noticed the mascot of the Czech National Bank. It is the only object that has this part of its panel highlighted together with Panel1.

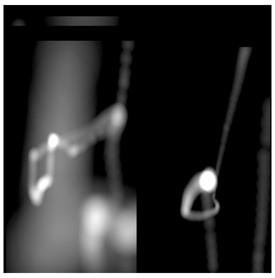
Panel2

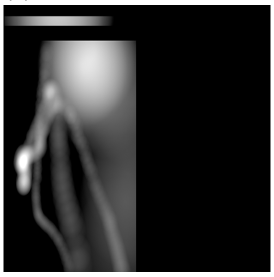
The panel with the highest trail density on its surface was Panel2, which has the advantage of being two-sided; it has graphics on both of its sides. It is located right among the first objects in the whole scene, next to the large CNB logo. Figures 6.24 and 6.25.

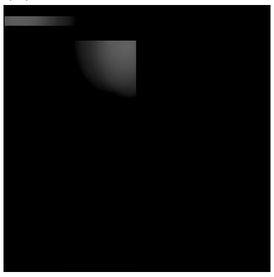

(a) Participant 1.

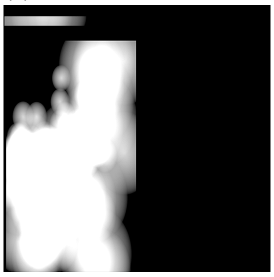

(b) Participant 2.

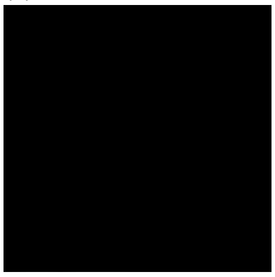

(c) Participant 3.

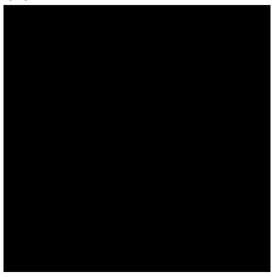

(d) Participant 4.

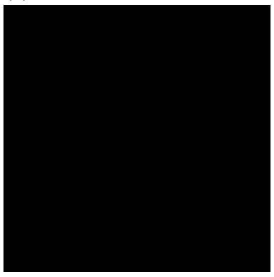

(e) Participant 5.

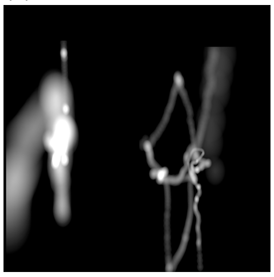

(f) Participant 6.


(g) Participant 7.


(h) Participant 8.


(i) Participant 9.


(j) Participant 10.


(k) Participant 11.

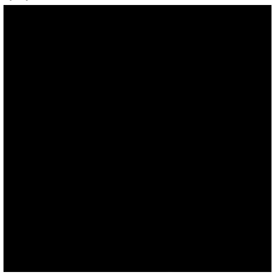
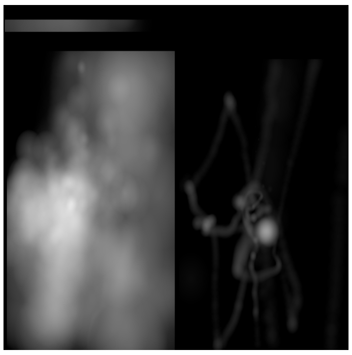
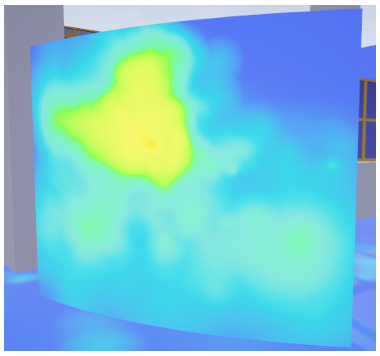
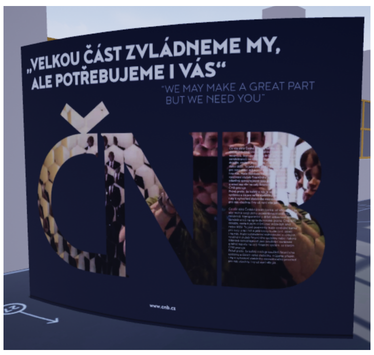

(l) Participant 12.

Figure 6.18 Heatmap texture variants of Panel11 object.

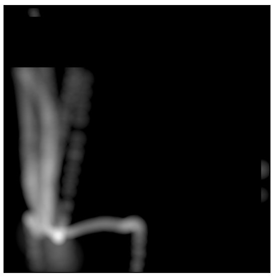

(a) Result gaze heatmap texture.

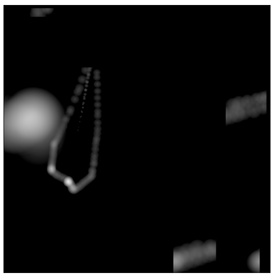

(b) Result gaze heatmap on object.



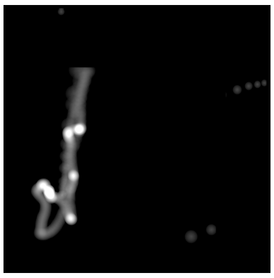
(c) Panel11 original material.

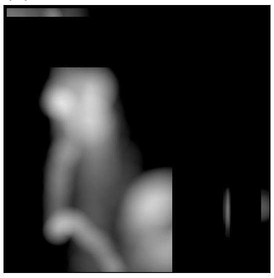
Figure 6.19 Resultant heatmap of Panel11.

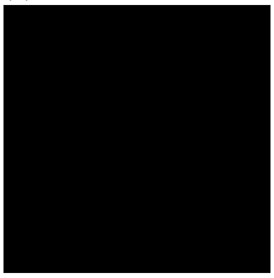

(a) Participant 1.

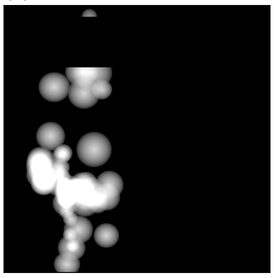

(b) Participant 2.

(c) Participant 3.

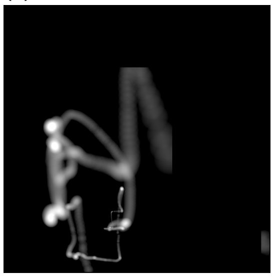

(d) Participant 4.

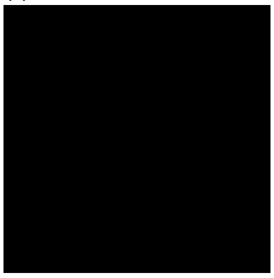

(e) Participant 5.

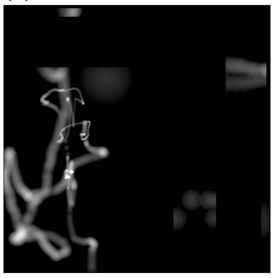

(f) Participant 6.


(g) Participant 7.

(h) Participant 8.


(i) Participant 9.


(j) Participant 10.


(k) Participant 11.

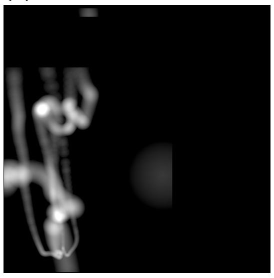
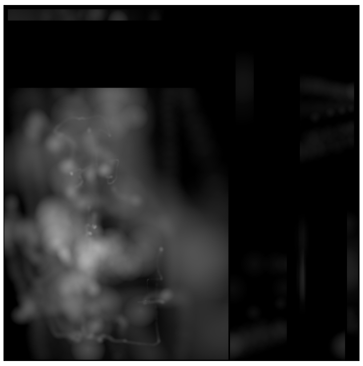
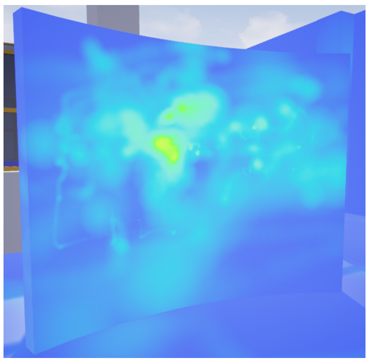
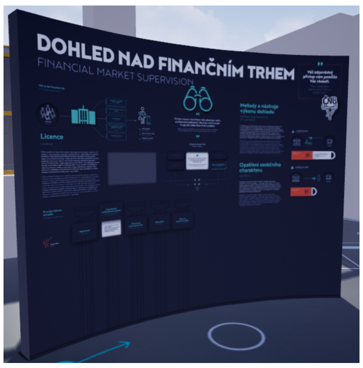

(l) Participant 12.

Figure 6.20 Heatmap texture variants of Panel14 object.

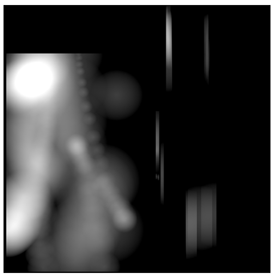

(a) Result gaze heatmap texture.


(b) Result gaze heatmap on object.



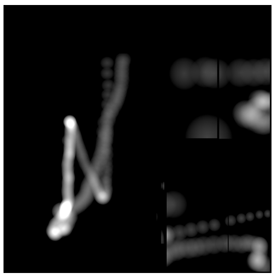
(c) Panel14 original material.

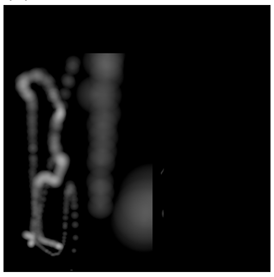
Figure 6.21 Resultant heatmap of Panel14.

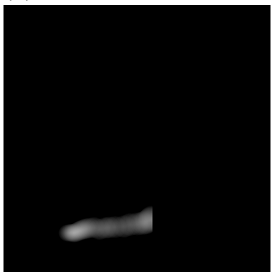

(a) Participant 1.

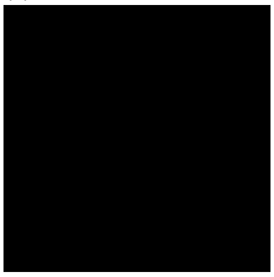
(b) Participant 2.

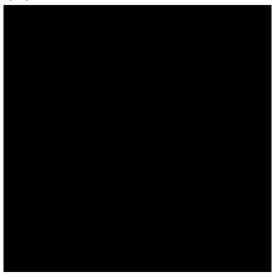
(c) Participant 3.

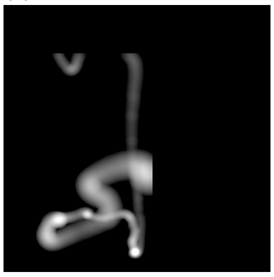

(d) Participant 4.


(e) Participant 5.

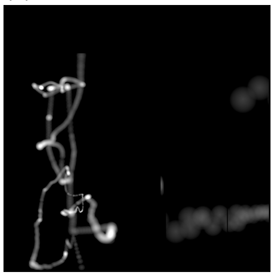

(f) Participant 6.


(g) Participant 7.


(h) Participant 8.


(i) Participant 9.

(j) Participant 10.


(k) Participant 11.

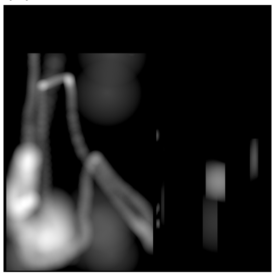
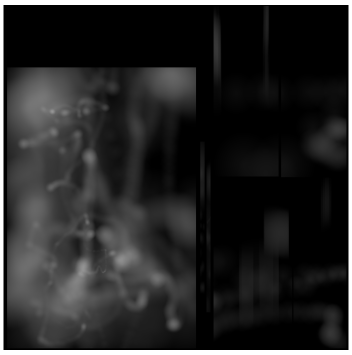
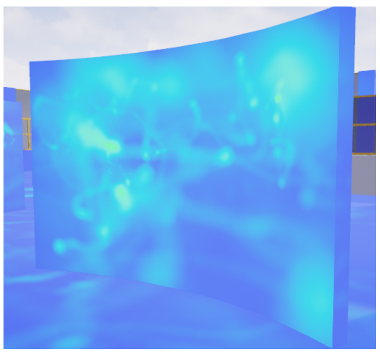
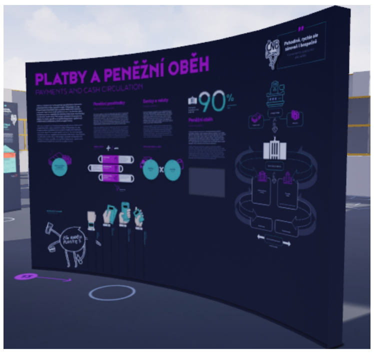

(l) Participant 12.

Figure 6.22 Heatmap texture variants of Panel17 object.


(a) Result gaze heatmap texture.

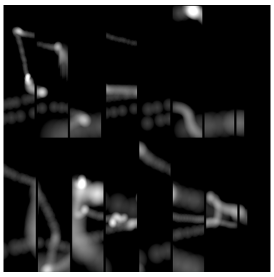

(b) Result gaze heatmap on object.

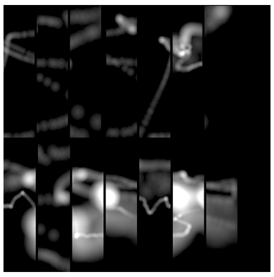


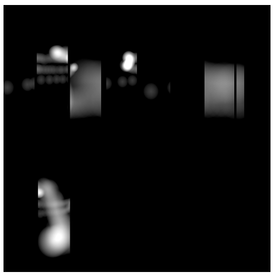
(c) Panel17 original material.

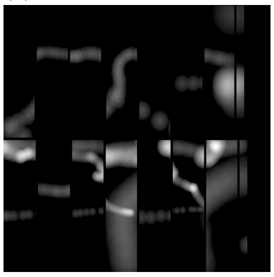
Figure 6.23 Resultant heatmap of Panel17.

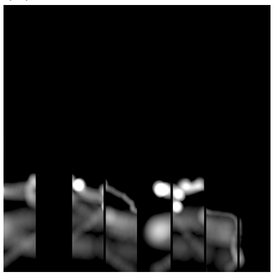
(a) Participant 1.

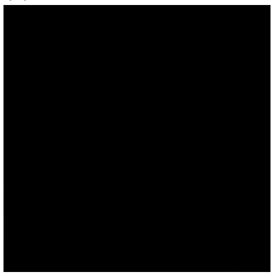

(b) Participant 2.

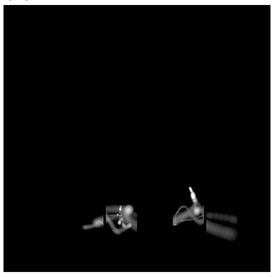

(c) Participant 3.

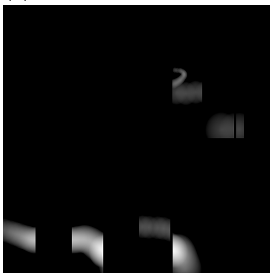

(d) Participant 4.

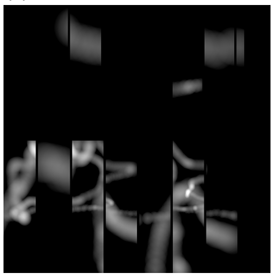

(e) Participant 5.

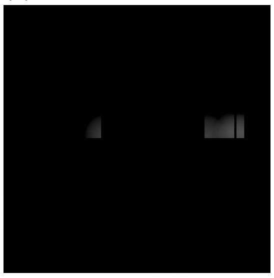

(f) Participant 6.


(g) Participant 7.


(h) Participant 8.


(i) Participant 9.


(j) Participant 10.


(k) Participant 11.

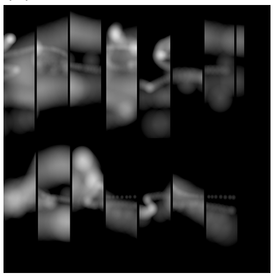
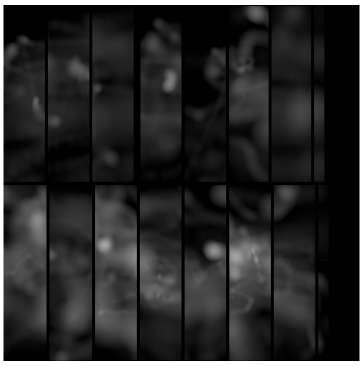
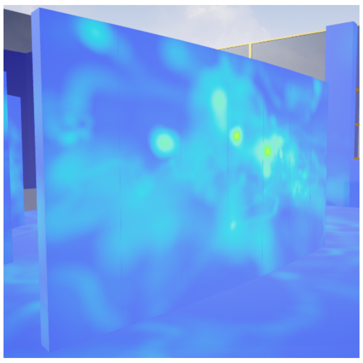
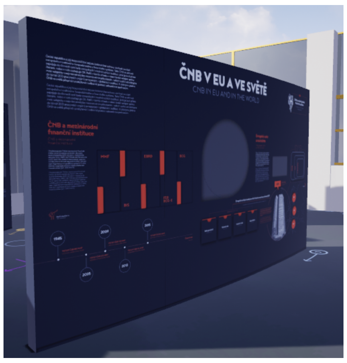

(l) Participant 12.

Figure 6.24 Heatmap texture variants of Panel2 object.


(a) Result gaze heatmap texture.


(b) Result gaze heatmap on object.



(c) Panel2 original material.

Figure 6.25 Resultant heatmap of Panel2.

Chapter 7

Conclusion

The main goal of this thesis was to develop a prototype application inside a modern game engine that collects and visualises gaze data and use it to build an experiment that collects data from multiple participants inside a 3D virtual scene.

An analysis of existing hardware and software solutions that use eye tracking was performed. Based on this analysis, it was decided that the prototype application would be designed as a plug-in for Unreal Engine 4. The plugin extends the functionality of the engine with the ability to create heatmaps in real-time using raycasting, which was designed to be independent of any specific ET hardware. The only thing the plug-in needs is a gaze origin and direction in world coordinates. The plug-in allows two types of heatmap to be captured at once. One that collects gaze data and one that collects forward vectors of a virtual camera. It is possible to visualise their mutual appearance.

An Unreal Engine scene was created from a prototype 3D model of the Czech National Bank Visitor Centre in Blender. It was originally intended to use the XTAL headset to collect gaze data, but due to the instability of its OpenXR runtime in Unreal Engine this could not be accomplished. Several unsuccessful attempts were made to fix the bug. Among them was the migration of the experiment project to the new Unreal Engine 5. Data collection from multiple participants was done with XTAL, but only using forward vectors of the virtual camera.

Heatmaps were collected from 12 different participants. These data were processed into resultant data that visualised the collective visual attention of this group of participants. From this, it was evaluated which objects in the scene were noticed the most and why.

Future Work

This thesis offers a working prototype of a plug-in for UE4 that collects and visualises gaze data using heatmaps that can be used to produce more complicated experiments exploring the behaviour of a much larger group of people than was used in this thesis. It is possible to try any other VR headset with ET. There is plenty of room for improvement. One of the main ideas that had not been done, because it was not even possible to test, is the classification of gaze data into saccades and fixations. Synchronise with the maximum output that the eye tracker offers and perform these classifications faster than each Tick. This could be used to create a scanpath or to store fixation metrics per region of interest in real time. This would allow for the implementation of other visualisation and data collection methods.

One way to extend it would be to implement an objective-based system similar to the one described in the analysis of Cognitive3D's software that would attach gaze events to objects that will get triggered when it is gazed at.

Appendix A
Unreal Engine 5 migration

The plug-in that was created in this work was simple to migrate to Unreal Engine 5. All that had to be done was to download the engine, create a new project, and then copy the previous `.uasset` files of the Blueprints and Materials in UE4 into the project. The entire experiment scene was copied into UE5 exactly the same way.

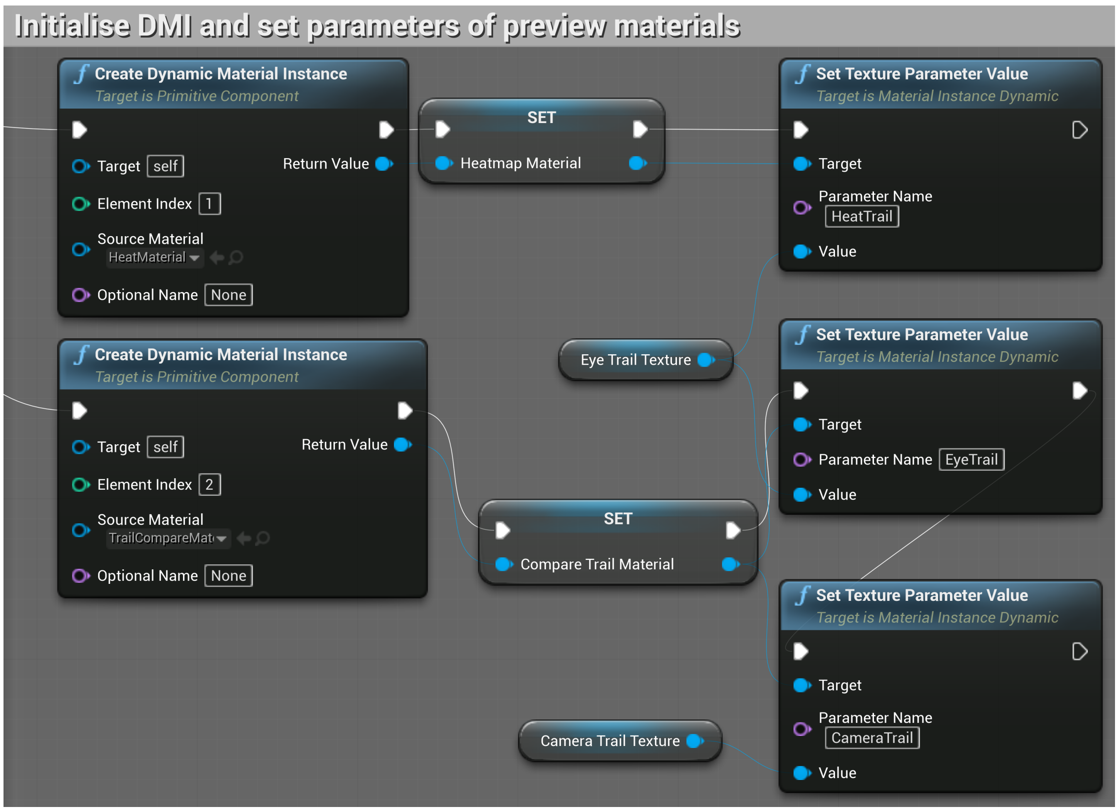
Immediately it was tested if the entire UEEditor would crash after running it with the XTAL headset, for normal scene traversal. This time it did not crash even once. However, another problem occurred during ET calibration in XTAL. Not a single calibration succeeded; all of them caused the UEEditor to crash while the OpenXR Eye Tracking plug-in was enabled.

This was not even consulted with support, as another issue was discovered that was not related to the headset. The technique of drawing heatmaps using a 3D brush stopped working. In the latest version of UE5, the Additive Composite mode of the Scene Capture Component does not work properly. In the official Unreal Engine forums, several users reported the same problem during UE5 beta testing [75]. The additive and composite mode of the Scene Capture Component behave the same as the overwrite mode. This bug was not fixed before submitting this thesis. Causes the heatmap texture to always be overwritten with a new brush according to the collision. This method could be used when the Epic Games team fixes the bug.

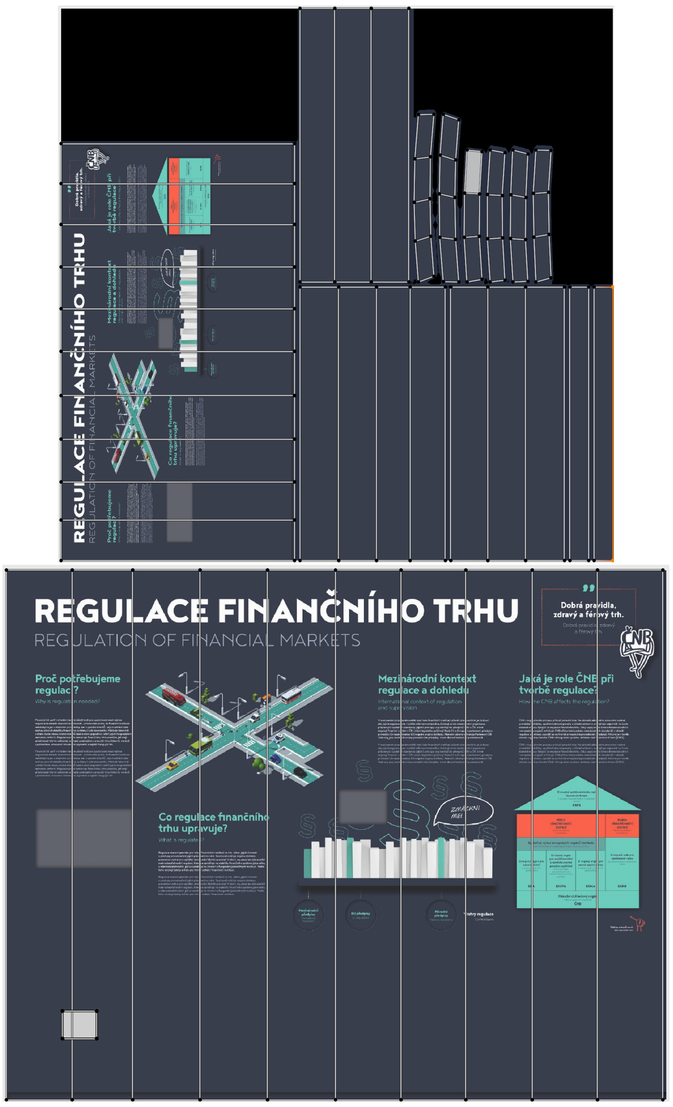
It was not possible to use Unreal Engine 5 to collect ET data with the prototype plug-in. The migrated UE5 version of the experiment scene can be found in the enclosed media in the `experiment/project/CNB_NC_ET_UE5.zip` file.

Appendix B

ET Component DMI initialisation


Initialise DMI and set parameters of preview materials
Create Dynamic Material Instance
Target is Primitive Component
Target self
Return Value
Element Index 1
Source Material
HeatMaterial
Optional Name None
SET
Heatmap Material
Set Texture Parameter Value
Target is Material Instance Dynamic
Target
Parameter Name
HeatTrail
Value
Create Dynamic Material Instance
Target is Primitive Component
Target self
Return Value
Element Index 2
Source Material
TrailCompareMat
Optional Name None
Eye Trail Texture
Set Texture Parameter Value
Target is Material Instance Dynamic
Target
Parameter Name EyeTrail
Value
SET
Compare Trail Material
Set Texture Parameter Value
Target is Material Instance Dynamic
Target
Parameter Name
CameraTrail
Value
Camera Trail Texture

Appendix C
Baked texture difference



REGULACE FINANČNÍHO TRHU
REGULATION OF FINANCIAL MARKETS
Proč potřebujeme regulaci?
Why is regulation needed?
Co regulace finančního trhu upravuje?
What is regulated
Mezinárodní kontext regulace a dohledu
International context of regulation and supervision
Jaká je role ČNB při tvorbě regulace?
How the ČNB affects the regulation?
Dobrá pravidla, zdravý a férový trh.

Bibliography

1. HANSEN, Dan Witzner; JI, Qiang. In the Eye of the Beholder: A Survey of Models for Eyes and Gaze. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2010, vol. 32, no. 3, pp. 478–500. Available from DOI: `10.1109/TPAMI.2009.30`.
2. DUCHOWSKI, Andrew T. Eye tracking methodology: theory and practice. In: 2nd ed. Springer, London, 2007, chap. 5, pp. 51–59. ISBN 978-1-84628-608-7.
3. COGNOLATO, Matteo; ATZORI, Manfredo; MÜLLER, Henning. Head-mounted eye gaze tracking devices: An overview of modern devices and recent advances. *Journal of Rehabilitation and Assistive Technologies Engineering*. 2018, vol. 5. Available from DOI: `10.1177/2055668318773991`.
4. MIND MEDIA. *EOG* [online]. Mind Media [visited on 2022-04-27]. Available from: `https://www.mindmedia.com/en/solutions/research/eog/`.
5. CHRONOS VISION. *Example of Scleral Contact Lens/Search Coil Eye movement measurement* [online]. Research Gate, 2018 [visited on 2022-04-27]. Available from: `https://www.researchgate.net/figure/Example-of-Scleral-Contact-Lens-Search-Coil-Eye-movement-measurement-Picture-courtesy_fig6_334988473`.
6. KAR, Anuradha; CORCORAN, Peter. A Review and Analysis of Eye-Gaze Estimation Systems, Algorithms and Performance Evaluation Methods in Consumer Platforms. *IEEE Access*. 2017, vol. 5, pp. 16495–16519. Available from DOI: `10.1109/ACCESS.2017.2735633`.
7. HANSEN, Dan Witzner; JI, Qiang. *Components of video-based eye detection and gaze tracking* [online]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2010 [visited on 2022-01-05]. Available from: `https://www.researchgate.net/publication/41028489_In_the_Eye_of_the_Beholder_A_Survey_of_Models_for_Eyes_and_Gaze/figures`.
8. TRNKA, Vladislav. *Eye Tracking in VR headset XTAL* [online]. 2020 [visited on 2022-04-23]. Available from: `https://dspace.cvut.cz/handle/10467/87746`. Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering. Supervised by Antonín POŠUSTA.
9. UGWITZ, Pavel. *Utilizing Eye-Tracking in Context of Virtual Reality* [online]. 2020 [visited on 2022-02-16]. Available from: `https://is.muni.cz/th/e2kqs/`. Master's thesis. Masaryk University, Faculty of Informatics, Brno. Supervised by Jiří CHMELÍK.
10. TOBII PRO. *How do Tobii Eye Trackers work?* [Online] [visited on 2022-05-30]. Available from: `https://www.tobiipro.com/learn-and-support/learn/eye-tracking-essentials/how-do-tobii-eye-trackers-work/`.
11. CLAY, Viviane; KÖNIG, Peter; KOENIG, Sabine. Eye Tracking in Virtual Reality. *Journal of Eye Movement Research*. 2019, vol. 12. Available from DOI: `10.16910/jemr.12.1.3`.

12. VIVE. *SingleEyeData Struct Reference* [online]. SRanipal Unreal Eye Documentation [visited on 2022-05-28]. Available from: `https://developer.vive.com/documents/914/Unreal_SRanipal_Eye_Documentation_5vNBies.zip`. detail of SingleEyeData Struct in Doxygen Documentation.
13. UGWITZ, Pavel; KVARDA, Ondřej; JUŘÍKOVÁ, Zuzana; ŠAŠINKA, Čeněk; TAMM, Sascha. Eye-Tracking in Interactive Virtual Environments: Implementation and Evaluation. *Applied Sciences*. 2022, vol. 12, p. 1027. Available from DOI: `10.3390/app12031027`.
14. KIT, Dmitry; KATZ, Leor; SULLIVAN, Brian; SNYDER, Kat; BALLARD, Dana; HAYHOE, Mary. Eye Movements, Visual Search and Scene Memory, in an Immersive Virtual Environment. *PLOS ONE*. 2014, vol. 9, no. 4, pp. 1–11. Available from DOI: `10.1371/journal.pone.0094362`.
15. COGNITIVE3D. *Dynamic Objects* [2022-06-08]. Cognitive3D Documentation [visited on 2022-06-08]. Available from: `https://docs.cognitive3d.com/dashboard/dynamic-objects/`.
16. HARTZBECH, Peter. *SMI and Apple: What It Means for the Future of Eye Tracking* [online]. iMotions, 2017 [visited on 2022-05-30]. Available from: `https://imotions.com/blog/smi-apple-eye-tracking/`.
17. TOBII. *Tobii Releases Eye Tracking VR Development Kit for HTC Vive* [online]. 2017 [visited on 2022-05-30]. Available from: `https://www.tobii.com/group/news-media/press-releases/2017/5/tobii-releases-eye-tracking-vr-development-kit-for-htc-vive/`.
18. TOBII. *The History of Tobii* [online]. Tobii, 2021 [visited on 2022-05-30]. Available from: `https://www.tobii.com/group/about/history-of-tobii/`.
19. TOBII PRO. *Dark and bright pupil tracking* [online] [visited on 2022-05-30]. Available from: `https://www.tobiipro.com/learn-and-support/learn/eye-tracking-essentials/what-is-dark-and-bright-pupil-tracking/`.
20. TOBII PRO. *Tobii Pro Glasses 3* [online] [visited on 2022-05-30]. Available from: `https://www.tobiipro.com/product-listing/tobii-pro-glasses-3/`.
21. TOBII. *Description of Tobii Glasses 3* [online] [visited on 2022-05-30]. Available from: `https://pbs.twimg.com/media/EZgdkt5X0AMMdjy?format=jpg&name=large`.
22. TOBII PRO. *Tobii Pro Fusion* [online] [visited on 2022-05-30]. Available from: `https://www.tobiipro.com/product-listing/fusion/`.
23. TOBII PRO. *Tobii Pro SDK documentation* [online] [visited on 2022-05-30]. Available from: `https://developer.tobiipro.com/`.
24. TOBII. *VIVE Pro Eye with Tobii* [online] [visited on 2022-06-02]. Available from: `https://vr.tobii.com/integrations/htc-vive-pro-eye/`.
25. TOBII. *HP Reverb G2 Omnicept Edition the future of XR* [online] [visited on 2022-06-02]. Available from: `https://vr.tobii.com/integrations/hp-reverb-g2-omnicept-edition/`.
26. TOBII. *Pico Neo 3 Pro Eye with native Tobii eye tracking* [online] [visited on 2022-06-02]. Available from: `https://vr.tobii.com/integrations/pico-neo-3-pro-eye/`.
27. TOBII. *Develop* [online]. Tobii XR Devzone [visited on 2022-06-02]. Available from: `https://vr.tobii.com/sdk/develop/`.
28. THE KHRONOS GROUP. *OpenXR Overview* [online]. 2019 [visited on 2022-06-02]. Available from: `https://www.khronos.org/openxr/`.
29. THE KHRONOS GROUP. *OpenXR Ecosystem Update* [online]. 2020 [visited on 2022-06-02]. Available from: `https://www.khronos.org/assets/uploads/apis/OpenXR-EcoSystem-Update_Jul20.pdf`.

30. THE KHRONOS GROUP. *OpenXR Solving XR fragmentation* [online] [visited on 2022-06-02]. Available from: https://www.khronos.org/assets/uploads/apis/OpenXR-After_4.png.

31. THE KHRONOS GROUP. *Coordinate system* [online]. The OpenXR Specification [visited on 2022-06-06]. Available from: https://www.khronos.org/registry/OpenXR/specs/1.0/html/xrspec.html%5C#coordinate-system.

32. MIETTINEN, Wili. *Industrial-Strength Eye Tracking in Varjo Headsets* [online]. Varjo Blog [visited on 2022-06-06]. Available from: https://varjo.com/blog/industrial-strength-eye-tracking-in-varjo/.

33. VARJO. *OpenXR* [online]. Varjo Developer [visited on 2022-06-06]. Available from: https://developer.varjo.com/docs/openxr/openxr.

34. VARJO. *High-resolution VR headset for professionals – Varjo VR-3* [online] [visited on 2022-06-06]. Available from: https://varjo.com/products/vr-3/.

35. VARJO. *Varjo Aero. Reach new heights in virtual reality* [online] [visited on 2022-06-06]. Available from: https://varjo.com/products/aero/.

36. VARJO. *Eye tracking with Native SDK* [online]. Varjo Developer [visited on 2022-06-06]. Available from: https://developer.varjo.com/docs/native/eye-tracking.

37. VARJO. *Varjo VR-3 Virtual Reality Headset* [online]. Varjo Store [visited on 2022-05-30]. Available from: https://store.varjo.com/Product%5C%20images/VR3.media.01.jpg.

38. VARJO. *Gaze data logging* [online]. Varjo Developer [visited on 2022-06-06]. Available from: https://developer.varjo.com/docs/get-started/gaze-data-collection.

39. VARJO. *Introducing Varjo XR-3, the only true mixed reality headset* [online] [visited on 2022-06-06]. Available from: https://varjo.com/products/xr-3/.

40. PALACKÝ UNIVERSITY OLOMOUC. *IVR (Immersive Virtual Reality) 3D projection system* [online]. 2021 [visited on 2022-04-24]. Available from: https://zakazky.upol.cz/contract_display_4269.html.

41. VRGINEERS. *XTAL 3 Virtual Reality* [online]. XTAL Shop [visited on 2022-06-06]. Available from: https://www.xtal.pro/product/xtal-3-vr.

42. VRGINEERS. *XTAL 8K* [online]. XTAL Shop [visited on 2022-06-06]. Available from: https://www.xtal.pro/product/xtal-8k-virtual-headset.

43. KOSTÍLEK, Milan. *Vrgineers support* [email]. 2022. private conversation.

44. VRGINEERS. *XTAL 3 Virtual Reality Headset* [online]. 2022 [visited on 2022-05-30]. Available from: https://vrgineers.com/wp-content/uploads/2022/01/xtal3_2nd-Edition_VR.png.

45. VRGINEERS. *Introducing the XTAL 3 — the world's most advanced virtual & mixed reality simulation headset* [online]. 2022 [visited on 2022-06-07]. Available from: https://vrgineers.com/introducing-the-xtal-3/.

46. VRGINEERS. *Reconfigurable Virtual & Mixed Reality Classroom Pilot Trainer* [online] [visited on 2022-06-07]. Available from: https://vrgineers.com/classroom-trainer/.

47. FME CTU. *New classrooms for learning in 3D and working with 3D objects* [online]. Faculty of Mechanical Engineering CTU in Prague, Aktuality, 2021 [visited on 2022-06-12]. Available from: https://www.fs.cvut.cz/aktuality/1824-212/nove-ucebny-pro-3d-vyuku-a-praci-s-3d-predmety/.

48. VRGINEERS. *Authentic cockpit* [online] [visited on 2022-05-30]. Available from: https://vrgineers.com/wp-content/uploads/2021/11/authentic-cockpit.png.

49. NISHINO, Hideaki. *First look: the headset design for PlayStation VR2* [online]. Playstation Blog, 2022 [visited on 2022-06-06]. Available from: `https://blog.playstation.com/2022/02/22/first-look-the-headset-design-for-playstation-vr2/`.

50. NISHINO, Hideaki. *Horizon Call of the Mountain from Guerrilla and Firesprite revealed for PS VR2* [online]. Playstation Blog, 2022 [visited on 2022-06-06]. Available from: `https://blog.playstation.com/2022/01/04/playstation-vr2-and-playstation-vr2-sense-controller-the-next-generation-of-vr-gaming-on-ps5/`.

51. COGNITIVE3D. *Build better immersive 3D experiences* [online] [visited on 2022-06-07]. Available from: `https://cognitive3d.com/`.

52. OSHAN, Harsev. *Cognitive3D Demo call* [meeting]. 2021. private zoom online meeting.

53. COGNITIVE3D. *Dashboard Concepts* [online]. Cognitive3D Documentation [visited on 2022-06-08]. Available from: `https://docs.cognitive3d.com/dashboard/concepts/`.

54. COGNITIVE3D. *SceneExplorer* [Cognitive3D Documentation] [visited on 2022-06-08]. Available from: `https://docs.cognitive3d.com/dashboard/scene-explorer/`.

55. COGNITIVE3D. *Gaze and Fixations* [online]. Cognitive3D Documentation [visited on 2022-06-08]. Available from: `https://docs.cognitive3d.com/unreal/gaze-fixations/`.

56. COGNITIVE3D. *Glossary of Metrics* [online]. Cognitive3D Documentation [visited on 2022-06-08]. Available from: `https://docs.cognitive3d.com/metrics-glossary/`.

57. COGNITIVE3D. *Objectives* [online]. Cognitive3D Documentation [visited on 2022-06-08]. Available from: `https://docs.cognitive3d.com/dashboard/objectives/`.

58. EPIC GAMES. *Unreal Engine* [comp. software]. Version 4.27.2 [released on 2021-08-19]. Available also from: `https://www.unrealengine.com/en-US/download`. Installing Epic Games Launcher to download different Unreal Engine versions is required.

59. EPIC GAMES. *Unreal Engine 4.27 Documentation* [online] [visited on 2022-06-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/`.

60. BINARYEGO. *Unreal Engine 4 – Blueprint Texture Painting* [online]. Youtube, 2016 [visited on 2022-06-14]. Available from: `https://www.youtube.com/watch?v=6BE2c5TBvV4`.

61. POTTER, Robert. *UE4 Tutorial – Make a Basic Heat Map* [online]. Youtube, 2017 [visited on 2022-06-14]. Available from: `https://youtu.be/QL51f8qdsm8?t=1590`.

62. BRUCKS, Ryan. *Automated and Improved UV Dilation* [online]. Shader Bits, 2016 [visited on 2022-04-23]. Available from: `https://shaderbits.com/blog/uv-dilation`.

63. TRAN, Tommy. *Dynamic Mesh Painting in Unreal Engine 4* [online]. Ray Wenderlich, 2018 [visited on 2022-04-23]. Available from: `https://www.raywenderlich.com/6817-dynamic-mesh-painting-in-unreal-engine-4`.

64. MEEKI GAMES. *UE4 Importing and Exporting Render Targets (Save and Load) using Blueprints* [online]. Youtube, 2021 [visited on 2022-06-15]. Available from: `https://youtu.be/bb_yw5Cvs6k`.

65. BOWMAN, James. *Python OpenEXR* [comp. software]. Excamera Labs. Version 1.2.0 [visited on 2022-06-20]. Available from: `https://www.excamera.com/sphinx/articles-openexr.html`.

66. OLIPHANT, Travis. *NumPy: A guide to NumPy* [online]. Trelgol Publishing, 2006– [visited on 2022-06-20]. Available from: `http://www.numpy.org/`.

67. BLENDER FOUNDATION. *Blender* [comp. software]. Version 3.1 [released on 2022-03-09]. Available also from: `https://www.blender.org/download/releases/3-1/`.

68. CZECH NATIONAL BANK. *Visitor Centre* [online]. 2022 [visited on 2022-06-15]. Available from: `https://nc.cnb.cz/`.

69. PROCHÁZKA, Vít; BURIÁNEK, Jan. *CNB Visitor Centre prototype* [3D model]. Blender, 2021. Provided by Ing. Jan Buriánek and Vít Procházka in cooperation with AVMEDIA SYSTEMS, a.s.. Created from the SGL Project for the Czech National Bank Visitor Centre.
70. BLENDER FOUNDATION. *Blender 3.1 Reference Manual* [online] [visited on 2022-06-19]. Available from: `https://docs.blender.org/manual/en/3.1/`.
71. RYAN KING ART. *Bake Textures From One UV Map to Another UV Map (Blender Tutorial)* [online]. Youtube, 2021 [visited on 2022-06-19]. Available from: `https://youtu.be/1HyexrUEIv0`.
72. VRGINEERS. *VRG OpenXR* [comp. software]. Beta Version 2.8 [released on 22-06-14]. Sent by support via email.
73. VRGINEERS. *VR Tool* [comp. software]. Vrgineers Portal. Version 2.5.1.65 [visited on 2022-06-20]. Available from: `https://portal.vrgineers.com/vr-tool/`. A login is required and an account can be created only through Vrgineers support.
74. VRGINEERS. *Plugin for Unreal Engine 4.27* [comp. software]. Vrgineers Portal. Version 2.06 [visited on 2022-06-20]. Available from: `https://portal.vrgineers.com/downloads/unreal-engine/`. A login is required and an account can be created only through Vrgineers support.
75. BORKS, Pete. *Scene Capture 2D - Additive composite mode not working on UE5* [online]. Unreal Engine Forums, 2021-10 [visited on 2022-06-21]. Available from: `https://forums.unrealengine.com/t/scene-capture-2d-additive-composite-mode-not-working-on-ue5/257257`.

Contents of the enclosed media

The contents are also available on faculty GitLab:
`https://gitlab.fit.cvut.cz/kvasnric/bp_kvasnric`.

- `experiment` — the directory with Unreal Engine project of the experiment
  - `data` — the directory with collected heatmap textures
    - `Participantdata.zip` — compressed zip archive of heatmap textures from participants
    - `ResultantData.zip` — compressed zip archive of resultant heatmap textures
  - `model` — the directory with Blender projects
    - `CNB-NC-original.blend` — Visitor Centre CNB 3D original model
    - `CNB-NC-modified.blend` — Visitor Centre CNB 3D model ready for export to UE
  - `project` — the directory with Visitor Centre CNB Unreal Engine projects
    - `CNB_NC_ET.zip` — zip archive with project in Unreal Engine 4
    - `CNB_NC_ET_UE5.zip` — zip archive with project in Unreal Engine 5
- `src` — the directory of source codes
  - `plug-in` — the Unreal Engine prototype plug-in source
  - `thesis` — the directory of LaTeX source codes of the thesis
  - `python` — the directory with Python source codes
    - `merger.py` — the EXR image merger script
- `text` — the thesis text directory
  - `thesis.pdf` — the thesis text in PDF format
- `video` — the directory with video demonstrations
  - `01-basic-painting-method.mp4` — H265 videofile
  - `02-3D-brush-problem.mp4` — H265 videofile
  - `03-simple-collision-only.mp4` — H265 videofile
  - `04-creating-heatmaps.mp4` — H265 videofile
  - `05-resultant-data-showcase.mp4` — H265 videofile
  - `06-participant4.mkv` — H265 videofile
  - `07-participant8.mkv` — H265 videofile