Master of Aerospace Engineering - Research Project

Mind-Gazing: Exploring the World through a Hybrid Brain-Computer Interface and Eye Tracker

---

Final Research Report

Due date of report: 27/03/2025

Actual submission date: 26/03/2025

Author(s):
Noelia Bascones González

Tutors:
Prof. F. Dehais
Dr. K. Cabrera Castillos

Starting date of project: 01/02/2024

Duration: 14 Months

Contents

List of Figures

Abstract

Purpose: This project explores how to use Brain-Computer Interface (BCI) technology in immersive Virtual Reality (VR) environments. The goal is to promote hands-free interaction by decoding user's intention via their brain responses

The goal is to allow users to interact without using their hands, by decoding visually evoked brain responses. Specifically, it focuses on a burst code-modulated Visual Evoked Potential (c-VEP) paradigm, which uses slow, pseudo-random flickers designed to improve both classification accuracy and user comfort. An additional goal is to minimize the calibration time required by typical BCI systems, reducing it to less than one minute.

Methods: We developed a modular system combining an Electro-encephalography (EEG) cap to capture brain signals, Timeflux for real-time brain signal acquisition, processing and decoding, and Unreal Engine to create the VR environment. Communication between components was handled using the Lab Streaming Layer (LSL). The classification was done using the *Ricker* model, a convolutional neural network implemented in Timeflux. Visual flickers (Ricker Patches) were added on top of VR objects using dynamic material overlays. Multiple groups of objects could flicker in parallel, each using its own burst c-VEP sequence. These sequences were selected and assigned dynamically during runtime, enabling flexible and adaptive stimulus control. The VR framework is structured around four main blueprints—`BP_Flickers Manager`,`BP_LSL`,`BP_Calibration` and `Game`-designed to ensure modularity and ease of integration into new game environments.

Results: A test game was created where the user focuses on one of three flickering cups. The system could successfully detect which object the user was looking at based only on their brain signals. Predictions were correct and updated in real time, with visual feedback shown inside the game.These results disclose that it is possible to use burst c-VEPs and real-time EEG decoding in virtual reality. The framework is flexible and can be used in other games or applications. This brings us closer to moving BCI technology outside the lab and into more natural, everyday environments.

Declaration of Authenticity

Please add this paragraph about plagiary in your pre-report. For more information about plagiarism, you can read for example http://www.ox.ac.uk/students/academic/goodpractice/about/

This assignment is entirely my own work. Quotations from literature are properly indicated with appropriated references in the text. All literature used in this piece of work is indicated in the bibliography placed at the end. I confirm that no sources have been used other than those stated.

I understand that plagiarism (copy without mentioning the reference) is a serious examinations offence that may result in disciplinary action being taken.

Date

24/06/2024

Signature

1 State of the Art

1.1 Reactive Brain-Computer Interfaces (rBCIs)

Reactive Brain-Computer Interfaces (rBCIs) enable users to interact with external systems by detecting and decoding involuntary brain responses elicited by sensory stimuli. These systems do not require the user to consciously generate specific brain patterns, but instead rely on passive observation of targeted stimuli to evoke neural responses that can be translated into commands.

The most common method used in rBCI systems is electroencephalography (EEG). EEG measures electrical activity from the brain via electrodes placed on the scalp, capturing the summed activity of populations of pyramidal neurons. It provides high temporal resolution (on the order of milliseconds), making it particularly well suited for detecting fast brain responses like event-related potentials (ERPs) or visual evoked potentials (VEPs) [1]. Recent advances in dry EEG and wireless systems have also enabled more comfortable and mobile BCI setups [2].

Figure 1: EEG cap

1.1.1 Common Reactive BCI Paradigms

P300. The P300 is a positive deflection in the EEG signal that appears approximately 300 ms after the presentation of a rare or significant stimulus. In the classical P300 speller [3], the user focuses on a letter within a matrix while rows and columns are flashed. When the target letter's row or column is illuminated, a P300 response is evoked, allowing classification. P300 systems are considered user-friendly and require relatively low calibration effort. However, they rely on multiple repetitions to ensure a good signal-to-noise ratio and can be slow in practice.

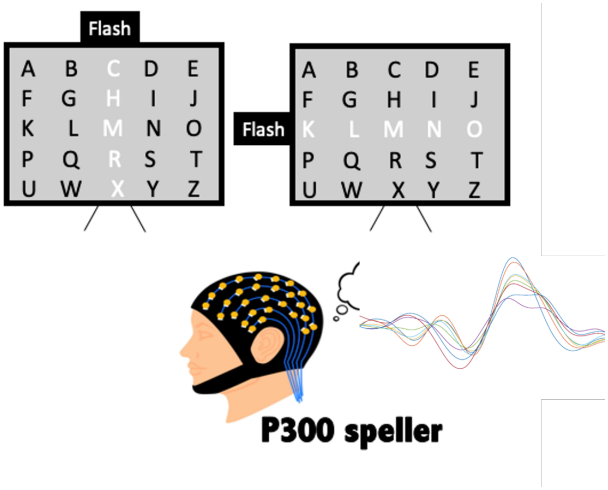

Figure 2: P300 speller

SSVEP. Steady-State Visually Evoked Potentials (SSVEP) are continuous oscillatory responses in the EEG, elicited when a user focuses on a flickering visual stimulus at a fixed frequency. Each command corresponds to a unique frequency. SSVEP-based BCIs offer high classification accuracy and scalability, but suffer from visual fatigue and may require up to 40 minutes of calibration, as it has to be done for each frequency [4, 5, 6].

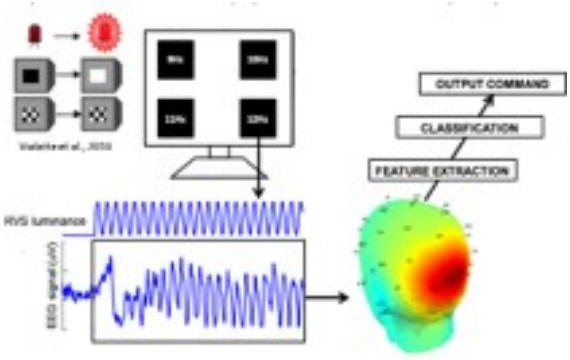

Figure 3: Steady State Visual Evoked Potential

Code-Modulated VEP (c-VEP). Code-modulated VEPs (c-VEPs) present a more recent alternative. Each stimulus is modulated using a unique pseudo-random binary sequence. The EEG synchronizes with the code of the attended stimulus, allowing decoding via correlation-based methods [7, 8, 9]. A key advantage is that all codes are phase-shifted variants of a master code. Only one needs to be learned, reducing calibration to under two minutes.

Burst Code-Modulated VEP (Burst c-VEP). This project implements a variation of the c-VEP paradigm known as burst c-VEP. Unlike standard c-VEP, where each onset is followed by several sustained frames, burst c-VEP uses short, transient onsets (1–2 frames) followed by immediate return to baseline. Onsets are spaced by at least 250 ms plus jitter, reducing overlap in EEG responses.

Although this reduces the number of available code sequences, the system retains the key benefits of c-VEP: low calibration, high accuracy, and reduced fatigue [10], while also getting better brain responses.

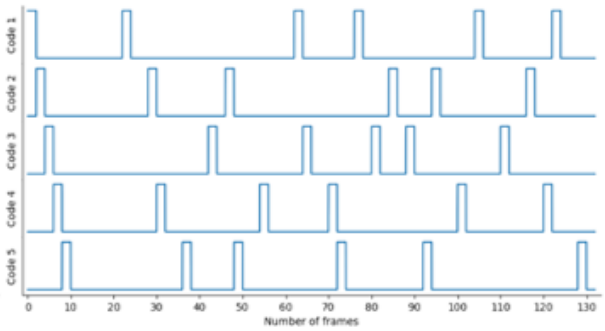

Figure 4: Burst c-VEP code sequences example

1.2 Design of Visual Stimuli: Ricker Patches

Traditional BCIs often rely on plain flickers—simple high-contrast flashes—to generate stimulus-locked brain responses. While effective in terms of neural detection, these stimuli frequently lead to visual discomfort, eye strain, and cognitive fatigue. This is especially problematic in multi-target setups where several flickering objects appear simultaneously in the user's peripheral vision, demanding high attentional effort to suppress distraction [2, 11].

To design more efficient and visually comfortable stimuli, we should consider how the brain processes visual information—visual neuroscience. This process begins in the retina, where two types of photoreceptors—cones and rods—convert light into neural signals. Cones, densely located in the fovea, are particularly well suited for detecting contrast and edges, responding strongly to localized changes in luminance and orientation [12, 13]. This makes it an ideal target for textured stimuli such as Ricker patches, which contain sharp boundaries and high contrast features.

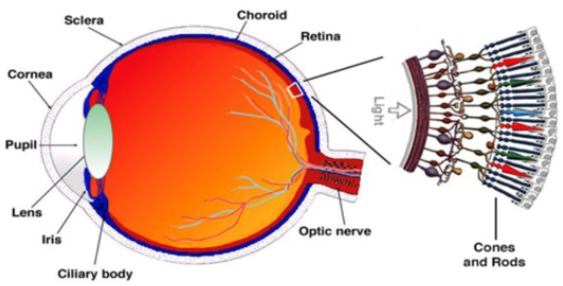

Figure 5: Foveal vision and central texture targeting

In contrast, the peripheral retina, dominated by rods, is far less sensitive to such spatial detail and is essentially "blind" to high-frequency visual patterns or fine texture,

especially when elements are densely clustered. This fundamental difference supports the use of small, high contrast textures placed centrally, where the fovea can extract them efficiently, while minimizing their saliency in the periphery [14, 11].

Signals are transmitted via the optic nerve to the primary visual cortex (V1) in the occipital lobe, where neurons are organized to respond selectively to orientation, frequency, and contrast [15].

The occipital region is the primary site for recording Visual Evoked Potentials (VEPs), which are brain responses time-locked to visual stimuli. In this project, VEPs are recorded using an 8-electrode dry EEG system, with sensors placed over occipital and parieto-occipital sites (PO7, PO3, POz, PO4, PO8, O1, Oz, O2). This configuration captures activity in V1 and surrounding areas, even with the reduced signal-to-noise ratio typical of dry EEG setups [2].

Despite the limitations of dry EEG, carefully designed stimuli can still elicit high-amplitude and coherent responses, sufficient for robust classification in a Burst c-VEP BCI system.

To address these limitations, this project employs Stimuli for Augmented Response (StAR): visual textures specifically designed to maximize foveal stimulation while minimizing peripheral visibility. In particular, we use Ricker wavelets (also known as Mexican hat functions), which are characterized by sharp central peaks surrounded by dark contours. These patches are ideal for stimulating ON-OFF retinal cells [16] and enhancing cortical responses.

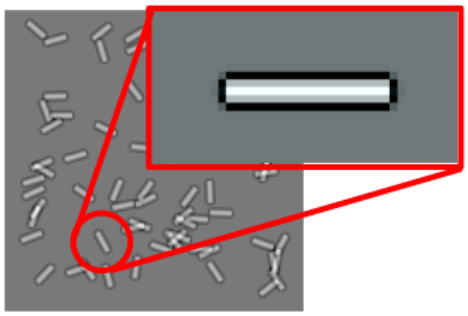

Figure 6: Ricker Patches

To generate the Ricker patches, we apply the following adapted version of the Ricker wavelet:

$$R(x) = \left(1 - \frac{x^4}{\sigma^2}\right) \exp\left(-\frac{x^4}{2\sigma^2}\right)$$

with $\sigma = 0.2$. This specific formulation was chosen to produce a flatter peak at $x = 0$, as opposed to a sharp spike [2]. This ensures that the central white portion of the flicker is wide enough to be visually prominent, while maintaining a thin black outline around it for edge definition. Once defined, the function is plotted in 2D and stretched to a fixed aspect ratio to create individual texture patches.

Each patch is small (less than 1° of visual angle), randomly oriented. The random

orientation of the patches is designed to engage a broader population of orientation-selective neurons in the primary visual cortex (V1) [15], while their small size and dense spatial arrangement limit peripheral perception through the *crowding effect* [11]. This design aims not only to boost decoding accuracy and reduce calibration time, but also to improve user experience, paving the way to push BCIs out of the lab and into practical, everyday environments.

1.3 BCI in Virtual Reality (VR) Environments

Integrating Brain-Computer Interface (BCI) systems into Virtual Reality (VR) environments has shown great potential to enhance user experience, increase immersion, and unlock new possibilities for natural brain-based interaction. Head-mounted displays (HMDs) used in VR provide several advantages for visual paradigms like P300 or neurofeedback (NF), as the stimuli are projected directly into the user's field of view, improving contrast and reducing environmental interference [17].

1.3.1 Examples of BCI-VR Integration

Kathner et al. [18] developed a P300-based speller in VR and compared it to a traditional monitor-based version. The results showed that VR achieved comparable online accuracy (96% vs. 95%), while also offering increased immersion and faster interaction.

Similarly, Cho et al. [19] explored the use of neurofeedback (NF) in a VR environment. Participants reported higher motivation and focus when using HMDs compared to standard displays, which were often perceived as tedious. This suggests that VR significantly enhances engagement and training quality in BCI applications.

Commercially, the company *Hybrid Black* has developed a Unity-based BCI system for VR. Their approach appears to use periodic and shifted sequences, likely based on modulated SSVEP paradigm [20].

1.3.2 Burst c-VEP and VR: Innovation in This Project

This project introduces a novel implementation of the **Burst c-VEP paradigm** within an immersive VR environment.

To our knowledge, this is the first implementation of burst c-VEP within VR, opening a new pathway for immersive, low-fatigue, reduce calibration times, high accuracy, and high-performance BCI applications. The implications of this research are far-reaching, with the potential to revolutionize sectors such as healthcare, aviation, and interactive entertainment.

2 System Installation, Setup, and Execution Pipeline

The system is composed of four fundamental components: the Enobio EEG cap, Timeflux, and Unreal Engine, Oculus Rift Headset. Each of these must be correctly installed and configured to ensure proper real-time communication and system performance within the VR environment.

In this the public GitHub repository[1] you will find the code for Timeflux, an example VR game (Shell Game) and the folder that has the framework to import on EVERY VR game that you develop.

SUPER IMPORTANT: The setup has to be in Windows, because the LSL plugin won´t work in Linux systems.

2.1 EEG Cap and NIC2 Application

To record EEG signals, we use the **Enobio®** device from Neuroelectrics. The management and configuration of this hardware is handled via the NIC2 software: **NIC2 (Neuroelectrics Instrument Controller)** provides a complete interface for device configuration, signal visualization, and streaming through the Lab Streaming Layer (LSL).[2]

2.2 Timeflux Installation and Launch

Timeflux is an open-source Python framework for real-time processing of biosignals. It must be installed along with its dependencies in a dedicated Python environment.

To begin, create a new conda environment and install the required packages:

```
conda create --name timeflux python=3.10 pip pytables
conda activate timeflux
pip install timeflux timeflux_ui timeflux_dsp pyriemann imblearn
timeflux -v
```

Next, clone the repository containing the project configuration:

```
git clone https://github.com/Noe6090/MindGazing-BCI-VR.git
```

Finally, to launch the Timeflux application (the folder of `RP_feedbackTimeflux`) with the appropriate pipeline, use the following command:

[1] `https://github.com/Noe6090/MindGazing-BCI-VR.git`
[2] `https://www.neuroelectrics.com/nic2`

```
conda activate timeflux
timeflux -d mainVR.yaml
```

The `mainVR.yaml` file (see Appendix A) contains the complete pipeline logic for EEG signal acquisition, calibration, classification, and real-time communication with the Unreal Engine VR environment.

2.3 Unreal Engine and Required Plugins

To install **Unreal Engine**, it is first necessary to download and install the Epic Games Launcher.[3] The recommended version of Unreal Engine is **5.3**, as it ensures compatibility with the required plugins:

- **Lab Streaming Layer Plugin:** Enables LSL integration for data exchange between Timeflux and Unreal.[4]
- **LE Extended Standard Library Plugin:** Provides extended functionality used within blueprints.[5]
- **OpenXR Plugin:** Allows Unreal Engine to interface with a wide range of VR headsets using the OpenXR standard. It ensures compatibility with different HMDs and provides access to advanced VR features.

IMPORTANT: Every time you create a new project, these plugins have to be enable.

2.4 VR Headset

The VR headset used in this project is the Oculus Rift. To enable its functionality on the development PC, it is necessary to download and install the official Oculus (Meta) software. This can be done by visiting the official setup page:[6].

Once the Meta app is installed, and the headset is properly connected to the laptop via the required cables, the Oculus Rift can be configured and linked through the application interface. This setup allows the headset to be recognized by Unreal Engine through the OpenXR plugin and used within the virtual environment.

[3] `https://store.epicgames.com/en-US/download`

[4] `https://www.fab.com/es-es/listings/67c60b96-90d1-4261-92a4-19a098a76c63`

[5] `https://www.fab.com/es-es/listings/0aadd41b-c02d-4f63-9009-bffad0070ebc`

[6] `https://www.meta.com/fr/quest/setup/?utm_source=dev.epicgames.com&utm_medium=oculusredirectc`

2.5 Project Setup in Unreal Engine

1. Create a new VR project.
2. Enable the Lab Streaming Layer, LE Extended Standard Library and OPenXR plugins.
3. Within the `Content` folder of the project, add the `Flickers_ToImport` folder.
4. Design the level as desired and tag the necessary actors following the structure described in the `Tagging System` .
5. Ensure the following Blueprints are added to the level, as only placed actors will be executed:
   - `BP_LSL`
   - `BP_FlickerManager`
   - `BP_Calibration`

2.6 Execution Pipeline

Once the system is installed and the level is built, the following steps must be followed to run the system:

1. Configure and set the Oculus Rift and its sensors
2. Make sure that the Meta App remains open
3. **Prepare the EEG cap:** Attach the Enobio EEG cap and ensure it is connected and streaming via NIC2.
4. **Launch Timeflux:** Open a terminal and execute the following command:

   ```
   timeflux -d .\mainVR.yaml
   ```

5. **Run the VR game:** Launch the Unreal Engine project. The system will wait for the code sequences from Timeflux before starting interaction.

Figure 7: Experimental setup

3 Unreal Engine - VR Framework

3.1 Flicker Integration in Unreal Engine

3.1.1 Framework Design Purpose

The BCI integration framework for Unreal Engine is designed to be modular and portable, allowing seamless integration into any Unreal Engine-based game. Its primary purpose is to enable the application of flickering visual stimuli on in-game objects, thus transforming any game into a Brain-Computer Interface (BCI) environment.

This framework acts as the core visual and logical interface between the brain signals processed by Timeflux and the visual feedback in the game. It provides all necessary tools to:

- Overlay flickering visual patches on any object in the scene.
- Assign a unique flicker sequence to each object or group of objects.

- Communicate in real time with Timeflux using Lab Streaming Layer (LSL).

The Unreal Engine framework is intended as a solid proof of concept. While future improvements are expected—such as optimization of logic, potential re-implementation in C++, and visual refinements—this version already offers a functional and extensible base to build upon.

The integration package consists of a folder that contains:

- **Blueprints:** Including `BP_Calibration`, `BP_FlickerManager`, `BP_LSL`, `BP_ChooseWhateverCup`, and `BP_CupsDynamic`.
- **Materials:** Including `M_Flicker` and associated texture assets.
- **Structures:** Custom Unreal Engine structs used for flickering logic and perceptual filtering: `Struct_ArrayMeshComponents`, `Struct_FlickerSequence`, and `Struct_PerceptionData`.

Core Blueprint Responsibilities:

The framework is primarily structured around three key Blueprints, each with a distinct and essential role:

- `BP_FlickerManager`: Contains all logic related to flicker behavior—initial setup, flicker assignment, and visibility control. See Figure 8.
- `BP_LSL`: Manages all communication between Unreal Engine and Timeflux using Lab Streaming Layer (LSL). It acts as the bridge for sending and receiving sequences, events, and predictions.
- `BP_Calibration`: Provides a template for performing calibration tasks using predefined flicker sequences. This Blueprint can be adapted or extended by the user to suit different experimental designs.

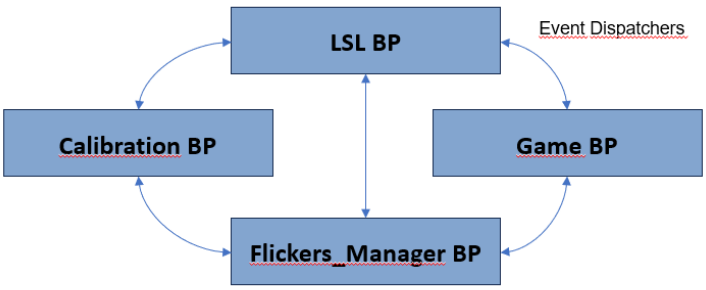

Figure 8: Main 4 Blueprints, and how they are connected through Event Dispatchers

While the calibration Blueprint is offered as an example and can be modified freely, it is strongly recommended to keep the `BP_FlickerManager` and `BP_LSL` as intact as possible. These are the backbone of the system, containing essential logic and communication structures. Any modification to them should be done with a thorough understanding of their internal mechanisms, as they serve as the "brain" of the BCI framework.

3.1.2 Method 1: HUD-based Approach (Deprecated)

Initially, a method based on a Heads-Up Display (HUD) overlay was explored. This approach projected flickering patch images on top of 2D projections of 3D objects using a `SceneCaptureComponent2D`. It worked by:

- Capturing a render of tagged objects onto a texture.
- Displaying that texture as a fullscreen widget overlay.
- Dynamically controlling the visibility of patches by updating the captured objects per frame.

While this method produced high-quality patches and handled scaling well, it was ultimately discarded due to several drawbacks:

- High computational cost, as the scene had to be re-captured every tick.
- Flickers from background objects could incorrectly occlude objects in the foreground.
- Increased complexity in managing overlap and occlusion logic.

Full technical details of this method can be found in Appendix C.

3.1.3 Method 2: Overlay Material Approach (Selected)

The second approach—and the one adopted in the final framework—relies on Unreal Engine's overlay material functionality. Instead of projecting flickers onto the screen, this method attaches a translucent overlay material directly onto each mesh component, giving the visual impression of patches placed directly on top of the objects' surfaces.

The key idea behind this method is that each flickering sequence received from Timeflux corresponds to a specific dynamic material instance. Instead of constantly creating and destroying materials as the flickering logic changes, the system creates one dynamic material instance per flicker code sequence. These instances are reused throughout the game and dynamically reassigned to the appropriate mesh components.

The pipeline works as follows:

1. A transparent texture containing the flickering patches is used to create a base overlay material (`M_Flicker`).
2. At runtime, for each code sequence received from Timeflux, the blueprint `BP_FlickerManager` creates a dynamic instance of this material and stores it inside a dedicated structure (`Struct_FlickerSequence`).
3. For each game cycle, the system dynamically determines which objects should flicker using perception-based filters (e.g., field of view, line of sight, and distance). This is handled by custom logic in the functions `SetPerceptionData` and `ArrangeIntoCodesStruct`.
4. Once each flicker sequence is associated with its corresponding objects, the system applies the relevant dynamic material instance as an overlay to those objects.
5. The visual flickering is achieved by toggling the `Visibility` scalar parameter of the overlay material. This value is updated according to the bits of the code sequence during gameplay.

This setup allows multiple objects to flicker simultaneously with different sequences. It is highly modular and scalable, as the system only requires as many dynamic material instances as flicker sequences. Assignments are updated dynamically as the player's viewpoint or scene conditions change.

The material itself is lightweight, consisting of a patch texture with adjustable parameters such as color, size, and brightness. Since the material is applied as an overlay and not a replacement, the original object textures remain fully visible beneath the patches. See Figure 9.

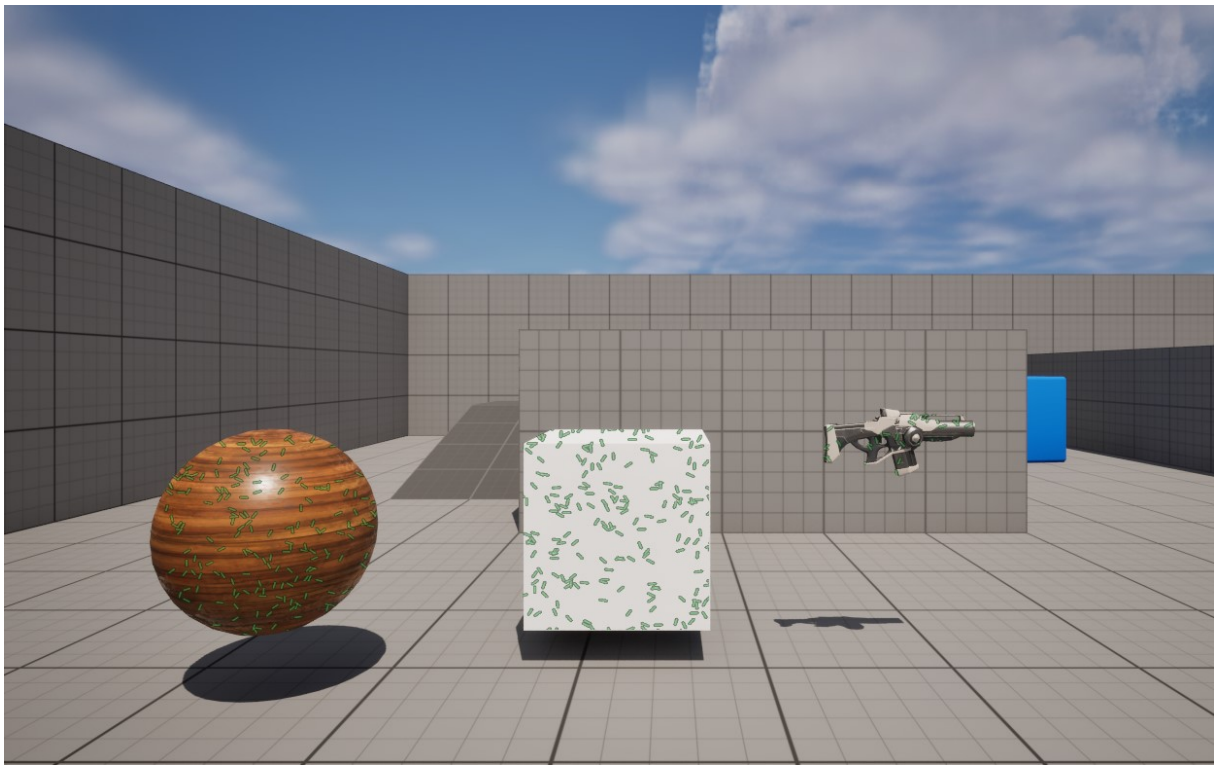

Figure 9: Ricker patches on top of different objects. Patches color is set to green and brightness to 0.4

3.2 Materials and Structures

This subsection documents the material and custom data structures used throughout the flicker system in Unreal Engine. These elements serve as fundamental building blocks and are essential for enabling dynamic visual stimulation and perceptual filtering.

3.2.1 M_Flicker Material

The material `M_Flicker` is used as a translucent overlay applied on top of mesh components to display the flickering patches. It is composed of a transparent texture containing the patch layout. See Figure 10.

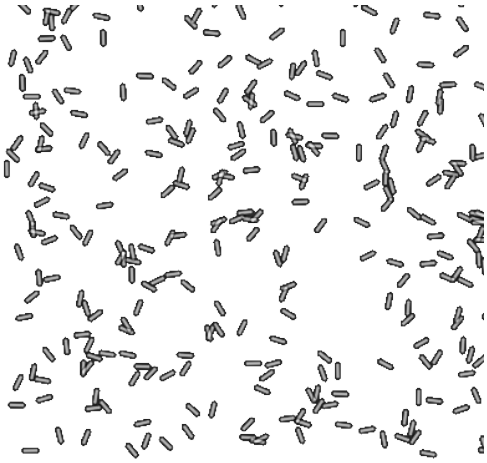

Figure 10: Ricker Patches

Configuration: See Figure 11.

- **Material Domain:** `Surface` – since the material is applied to object surfaces.
- **Blend Mode:** `Translucent` – to ensure the patch texture renders transparently above the base object material.
- **Shading Model:** `Default Lit` – selected after testing different shading models in VR to maximize patch visibility and contrast.

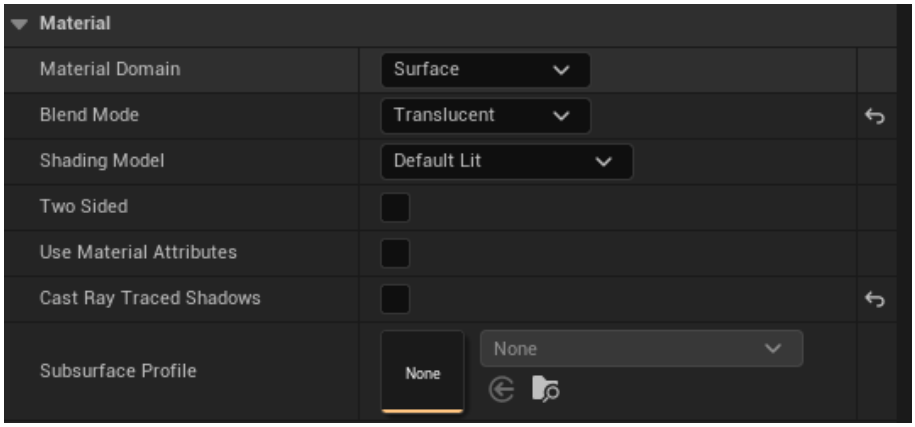

Figure 11: M_Flicker Configuration

Exposed Parameters:

- `SizePatches`**:** Controls the scale of the flickering patches.
- `ColorFlicker`**:** Defines the RGB color of the flickers.
- `Brightness`**:** Modifies patch intensity to adjust amplitude depth (e.g., 30% visibility instead of 100%).

- `Visibility`: A scalar value dynamically updated during runtime (see `BP_FlickerManager`) to show or hide the flicker.

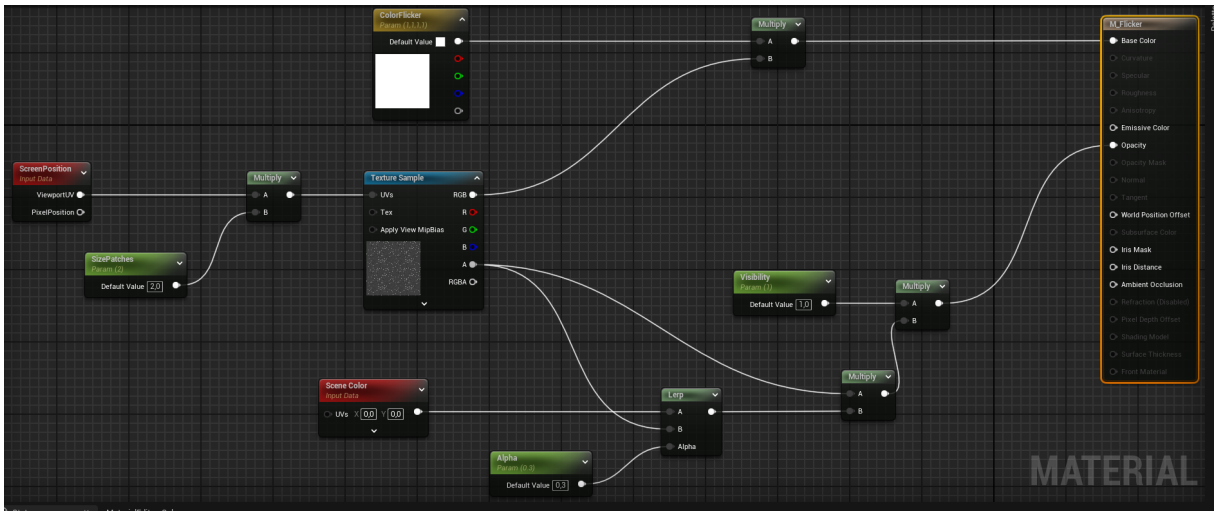

Figure 12: M_Flicker design graph

Dynamic material instances of `M_Flicker` are created at runtime by `BP_FlickerManager` and stored within each `Struct_FlickerSequence`. These instances are then applied as overlay materials on objects chosen to flicker during each cycle.

3.2.2 Struct_ArrayMeshComponents

Purpose: Stores arrays of mesh components to allow grouping and batch processing (e.g., applying flickers to multiple objects together).

Fields:

- `ArrayMeshComponents`: An array of `StaticMeshComponent` elements.

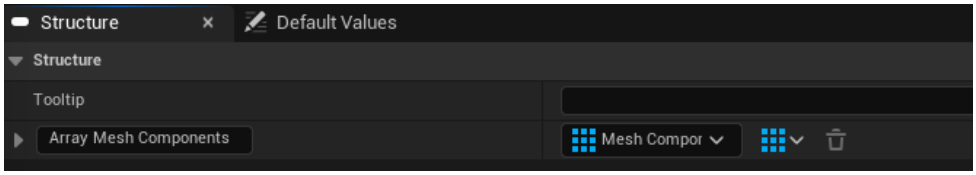

Figure 13: Struct_ArrayMeshComponents

This struct is mainly used to pass sets of components between internal functions, such as in dynamic assignment handled by `ArrangeIntoCodesStruct`, or `GetMeshComponentsByTag`.

3.2.3 Struct_FlickerSequence

Purpose: Represents a single flicker sequence and the necessary data to manage it during gameplay.

Fields:

- `SequenceID (int):` Unique identifier for the sequence.
- `Sequence (Array<int>):` Bit pattern for flickering (e.g., [1, 0, 1, 1, 0, ...]).
- `DynamicMaterial (MaterialInstance):` A dynamic instance of `M_Flicker`.
- `AssociatedMeshComponents (Array):` Objects currently assigned to this sequence.

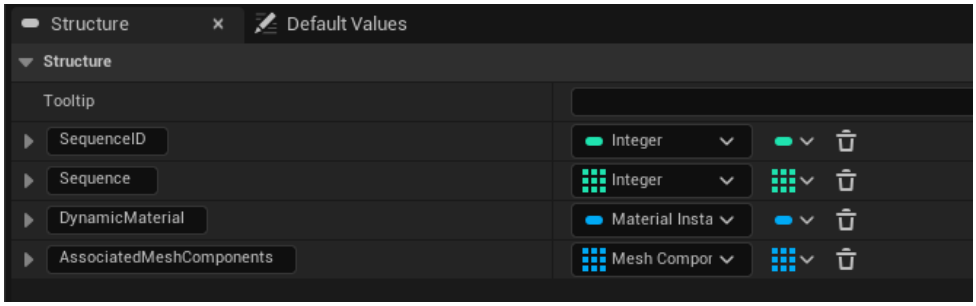

Figure 14: Struct_FlickerSequence

These structures are created in the function `ParseStringToFlickerSequenceStruct` and stored within an array (`ArrStruct_FlickerCodeSequences`) inside `BP_FlickerManager`.

3.2.4 Struct_PerceptionData

Purpose: Stores perceptual filtering information for mesh components. This allows real-time decisions on whether an object is a valid flickering candidate.

Fields:

- `StaticMeshComponents (Array):` Meshes being evaluated.
- `FOV (bool):` True if the component is within the player's field of view.
- `InSight (bool):` True if the object is not totally occluded (line-of-sight check passed).
- `Distance (float):` Distance from the player to the object.

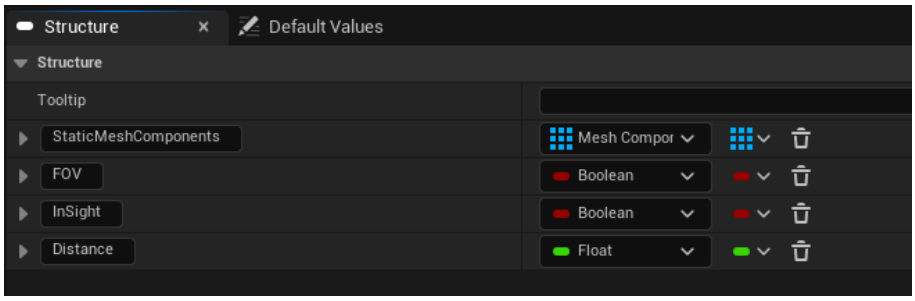

Figure 15: Struct_PerceptionData

This struct is filled and updated inside `SetPerceptionData`, and used by the `BP_FlickerManager` to choose which objects can be assigned to a flicker sequence during runtime.

If any one of the grouped mesh components passes the filters, all associated components in the group will be considered valid for flickering.

3.3 Tagging System

In order to determine which objects should receive flickering overlays, the system uses a custom tag-based mechanism. Tags are applied at both the actor level and the component level. This tagging strategy allows for grouping components, assigning code sequences, and differentiating behavior across game stages (e.g., calibration vs. gameplay).

3.3.1 Actor and Component Tag Rules

Each actor in Unreal Engine can have multiple tags, such as `"Calibration"` or `"Flicker"`. Likewise, each component within an actor can also have its own tags, represented as integers. The tag at index i in the actor's tag list is paired with the tag at index i in each component's tag list.

The key rule is: If an actor has a tag at index i, only the component tag at index i will be considered to determine grouping under that actor tag.

For example, consider an actor with the following:

- Actor tags: `["Calibration", "Flicker"]`
- Component `A1_C1` has component tags `[0, 2]`

Then:

- `Calibration` is at index 0 → the component will be grouped using tag 0

- `Flicker` is at index 1 → the component will be grouped using tag 2

This tagging system allows a single actor to be involved in different flickering contexts using independent grouping rules for each.

Additional rules:

- If a component lacks a tag at a specific index:
  - If other components have valid tags → it is excluded from the group.
  - If none of the components have valid tags → all components are grouped together by default.
- Two different actors will never share the same flicker code sequence, even if their tags and tag components are identical.

3.3.2 Tag Indexing and Grouping

This tag system is critical to the grouping logic used in the function `GetMeshComponentsByTag`. The function parses each actor's tag-component pairing and groups components into arrays of `Struct_PerceptionData`, which are later passed to filtering functions such as `SetPerceptionData` and assigned via `ArrangeIntoCodesStruct`.

Example:

Imagine the following actors and components:

Actor A1: Tags: `["Calibration", "Flicker"]` Components:

- `A1_C1`: tags `[0, 2]`
- `A1_C2`: tags `[0, 1]`
- `A1_C3`: tags `[1, None]`

Grouping for "Calibration" (tag index 0):

- Group 1: { A1_C1, A1_C2 } — same tag component (0)
- Group 2: { A1_C3 } — different component tag (1)

Grouping for "Flicker" (tag index 1):

- { A1_C1 } — component tag = 2

- { A1_C2 } — component tag = 1
- A1_C3 is excluded (tag = None)

Actor A2: Tags: `["Flicker", "Calibration"]` Components:

- `A2_C1`: tags `[0, None]`
- `A2_C2`: tags `[2, None]`
- `A2_C3`: tags `[1, None]`

Grouping for "Flicker" (tag index 0):

- { A2_C1 }, { A2_C2 }, { A2_C3 } — all grouped separately

Grouping for "Calibration" (tag index 1):

- { A2_C1, A2_C2, A2_C3 } — all component tags are None → grouped together by default

Calling the event `SetUpFlickers` with:

- Tag = `"Calibration"` → groupings:
  - { A1_C1, A1_C2 }, { A1_C3 }, { A2_C1, A2_C2, A2_C3 }
- Tag = `"Flicker"` → groupings:
  - { A1_C1 }, { A1_C2 }, { A2_C1 }, { A2_C2 }, { A2_C3 }

If there are more flicker candidates than available code sequences, dynamic filtering will occur based on perception conditions (FOV, visibility, distance), using `GetChosenComponentsToFlicker` and `ArrangeIntoCodesStruct`.

3.4 FlickerManager Blueprint – Internal Logic

3.4.1 General Blueprint Architecture

The `BP_FlickerManager` is a class actor blueprint and serves as the core module responsible for configuring, assigning, and updating all flickering behavior in the system. It centralizes the main logic and acts as a hub that interacts with other components such as `M_Flicker`, dynamic material instances, sequence data structures, and perception filters.

This blueprint is composed of the following main elements:

- **Event Graph:** The main execution flow where all high-level logic related to the flicker system is triggered and managed. It acts as the coordination layer, connecting input events, function calls, data updates, and dispatchers.
- **Functions:** Encapsulated logic for tasks such as parsing sequences, creating materials, filtering components by perception, and updating flickers.
- **Macros:** Reusable logic snippets used for formatting tags or batching operations.
- **Variables:** Blueprint-level data used to store settings, arrays of structures, timers, material instances, and references to tagged actors/components.
- **Event Dispatchers:** Used to broadcast state changes (e.g., `Epoch`, `CodesChanged`) to other blueprints such as `BP_LSL` or `BP_Calibration`.
- **Components:** Optional visual or functional components (e.g., scene anchors or debug meshes) used for spatial logic or visualization. In this case, this blueprint doesn't contain any visual component.

This modular organization allows the system to remain scalable and adaptable. Each block of logic is clearly isolated into specific functions or macros, making it easier to understand, maintain, and extend.

Special care has been taken to ensure that this blueprint only manages flicker-related behavior. Other tasks such as LSL communication or gameplay logic are delegated to `BP_LSL` or to the specific game blueprints (see `BP_ChooseWhateverCup`, `BP_CupsDynamic`).

3.4.2 Event SetUpFlickers

`SetUpFlickers(Tag, CodeSequencesString, PatchesVisible)` configures the flickering components for a specific group.

- Extracts all mesh components with the specified input tag using `GetMeshComponentsByTag`.
- Parses the code sequence string received from Timeflux to retrieve each individual sequence and its bits using `ParseStringToFlickerSequenceStruct`.
- Creates a new array of structures, one for each code sequence, storing sequence data and the associated dynamic material.
- Applies dynamic material instances to the corresponding mesh components.
- Optionally sets the visibility of flickers at the start based on the `PatchesVisible` boolean.

Figure 16: Event SetUpFlickers

3.4.3 Event DynamicChoosingComponent

This event dynamically evaluates which objects should be associated with each code sequence based on real-time perception filters:

- Calls `SetPerceptionData` to check which objects are visible, within line of sight, and within an acceptable distance.
- Calls `ArrangeIntoCodesStruct` to update object-to-sequence assignments accordingly.

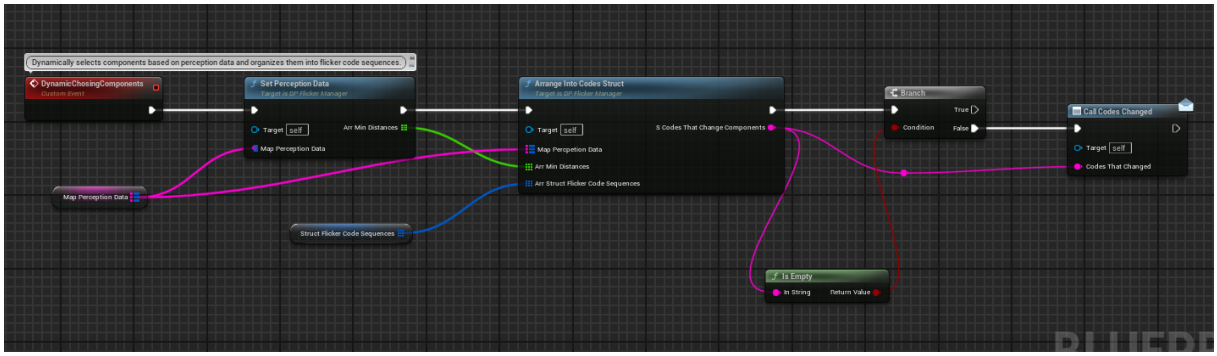

Figure 17: Event DynamicChoosingComponent

If a code sequence was previously associated with one object, and due to the updated perception another object is now more suitable, the assignment is changed. When this happens:

- The ID of the modified sequence is added to the list of codes with changed components.
- The dispatcher `CodesChanged` is triggered.
- `BP_LSL` listens to this and sends the update to Timeflux, allowing Timeflux to reset the accumulation and update its prediction logic accordingly.

3.4.4 Event UpdateBlink

This event keeps the flickering pattern synchronized over time. It is called on every flickering cycle (e.g., every 0.016667 seconds, equivalent to 60 Hz).

- All code sequences advance one bit simultaneously.
- The visibility of patches is updated according to the current bit.
- The event dispatcher `Epoch` is triggered with the appropriate information:
  - During calibration: sends the actual bit (since only one object flickers).
  - During gameplay: sends the bit index instead, because each sequence has a different bit at each step. Timeflux already knows the sequences and can match the index to the correct bit.

Important: The LSL stream that sends the `Epoch` event to Timeflux must follow a two-column format:

- `label`: string ("epoch")
- `data`: integer (either bit or index)

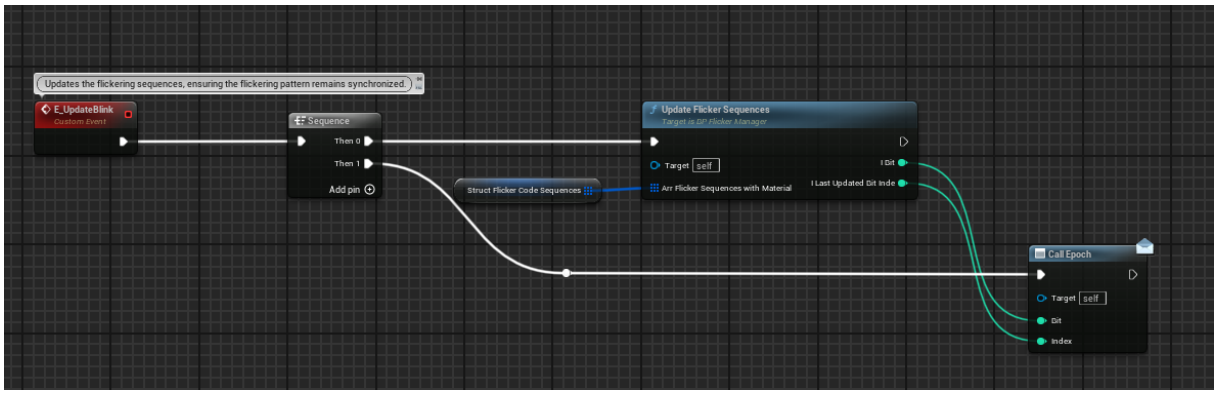

Figure 18: Event UpdateBlink

3.4.5 Event FlickerEverythingWithThisTag

Starts the flickering process for all components with a given tag. Inputs are: `Tag`, `CodeSequencesString`, `DoSetupFlickers` (boolean).

`DoSetupFlickers` controls whether `SetUpFlickers` should be called.

Afterwards, two timers are set using "Set Timer by Event":

1. **Perception Timer:** Interval = perception update rate (in seconds). → Triggers `DynamicChoosingComponent()`.
2. **Flicker Timer:** Interval = flicker rate (e.g., 0.016667 s = 60 Hz). → Triggers `UpdateBlink()`.

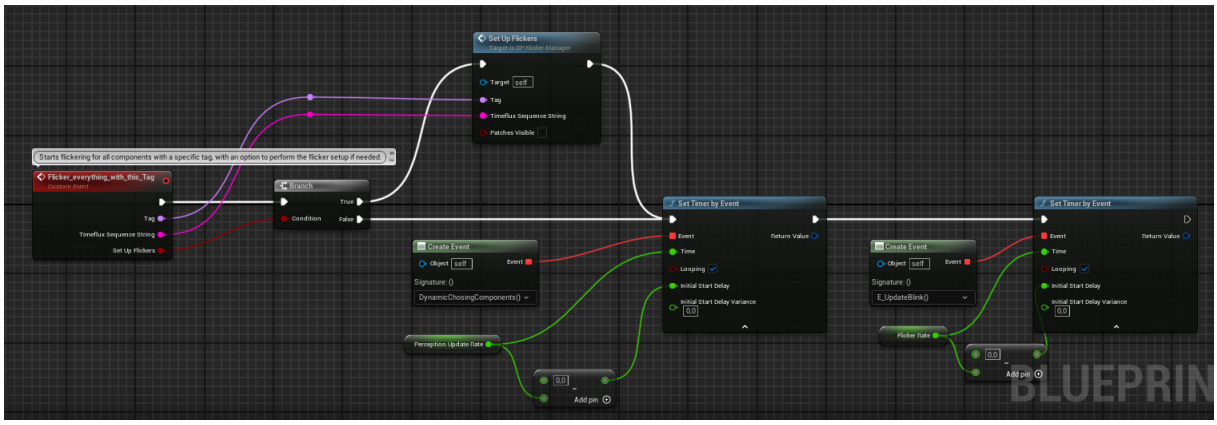

Figure 19: Event FlickerEverythingWithThisTag

3.4.6 Event ResetCodeSequences

Fully clears the array of `Struct_FlickerSequence`. This is used to completely reset the flicker state, allowing a new flicker setup to be initialized from scratch.

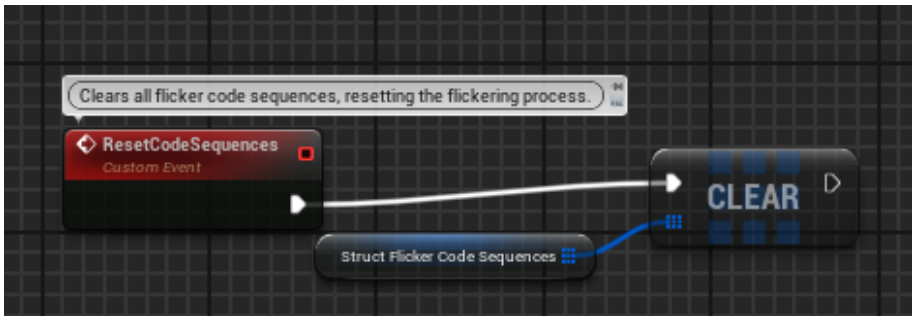

Figure 20: Event ResetCodeSequences

3.4.7 Event StopFlickering

Stops all flickering activity. Inputs: `ResetIndexCode (bool)`, `ResetCodeSequences (bool)`.

Steps:

1. Clears both flicker timers (UpdateBlink and DynamicChoosingComponent).

2. Resets the overlay materials of any objects that currently had flickers.
3. If `ResetIndexCode = true`, sets the bit index back to 0.
4. If `ResetCodeSequences = true`, calls `ResetCodeSequences()`.

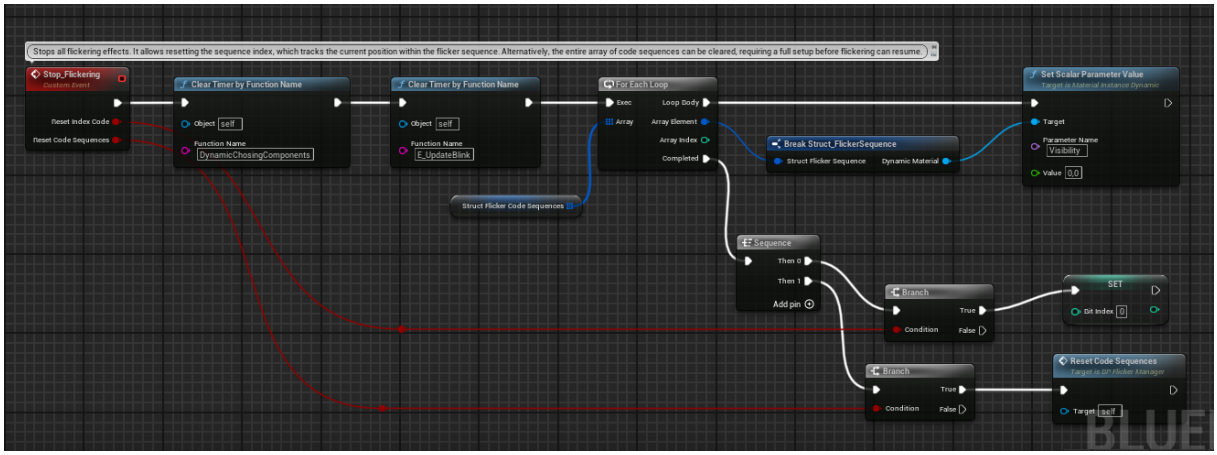

Figure 21: Event StopFlickering

3.4.8 Event PauseFlickering

Temporarily stops flickering for a specified duration without clearing the setup. Inputs: `TimePaused (float)`, `ResetIndexCode (bool)`.

What this event does:

1. It first calls the event `StopFlickering`, but with the boolean `ResetCodeSequences` set to `false`. This is very important, because we do **not** want to clear the array of flicker code sequences — otherwise, we would have to perform the whole flicker setup again, and that is not desired in this case.
2. After waiting for the time specified by `TimePaused`, the system calls the event `FlickerEverythingWithThisTag`. At this point, it doesn't matter which `Tag` or `CodeSequencesString` are passed as input, because the event `SetUpFlickers` inside will not be called.
3. This is because we explicitly set the `DoSetupFlickers` boolean to `false`, as we want to resume flickering exactly as it was before the pause — without triggering a new setup.

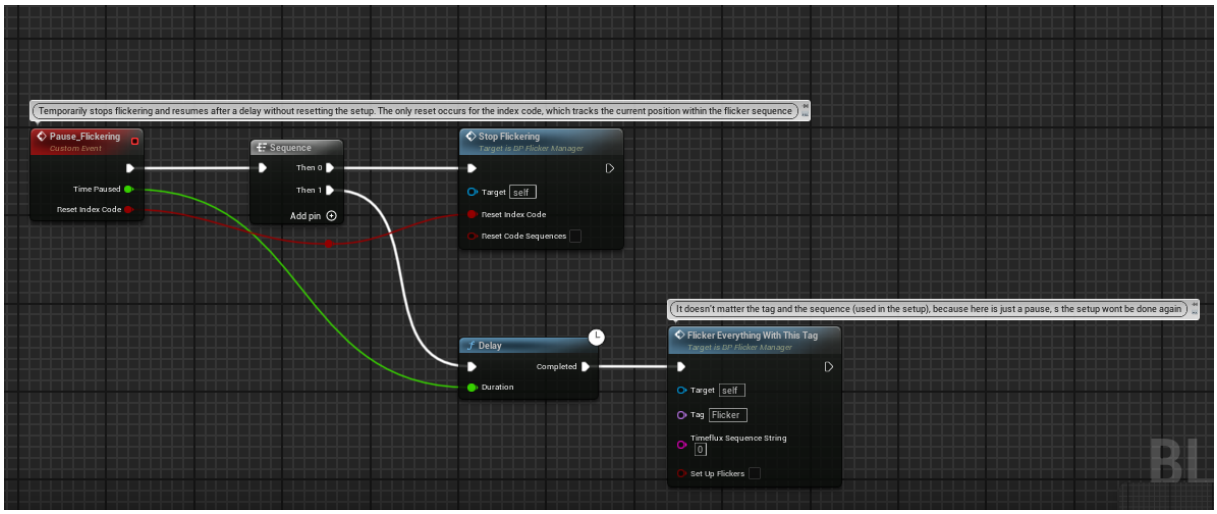

Figure 22: Event PauseFlickering

3.4.9 Function StringSequenceToArrayInt

This function takes a text sequence and converts it into an array of numeric values. It filters out all non-numeric characters and keeps only the digits, returning them as integers in an array.

For example, the input string `["1011"]` would result in the array `[1, 0, 1, 1]`.

This function is mainly used inside `ParseStringToFlickerSequenceStruct` to convert the raw sequence string received from Timeflux into usable bit arrays.

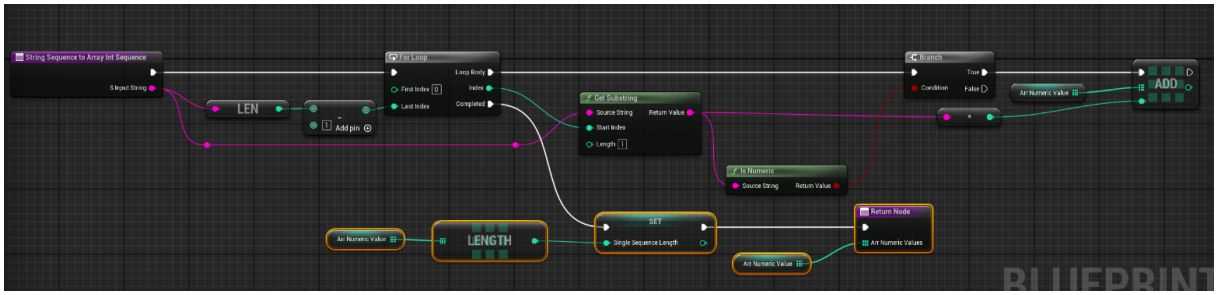

Figure 23: Function StringSequenceToArrayInt graph

3.4.10 Function ParseStringToFlickerSequenceStruct

This function receives a single string containing all the flicker sequences sent by Timeflux. If Timeflux sends, for example, two sequences, they arrive as one concatenated string:

```
['110000...001100', '001100...000000']
```

What this function does:

1. It splits the string using commas and separates each individual sequence, even if brackets and quotes are included.

2. Then, for each sequence, it calls `StringSequenceToArrayInt` to keep only the numeric characters and convert them into an array of integers (bits). For example, two arrays of 132 bits each.

3. For each resulting bit array, it creates a new `Struct_FlickerSequence`, assigns a unique `SequenceID`, and stores the bits for later use.

These structures will be essential for assigning objects dynamically using the flicker system, and will later be extended using other functions such as `UpdateFlickerSequences`, `ArrangeIntoCodesStruct`, and `AssignDynamicMaterialToFlickerSequence`.

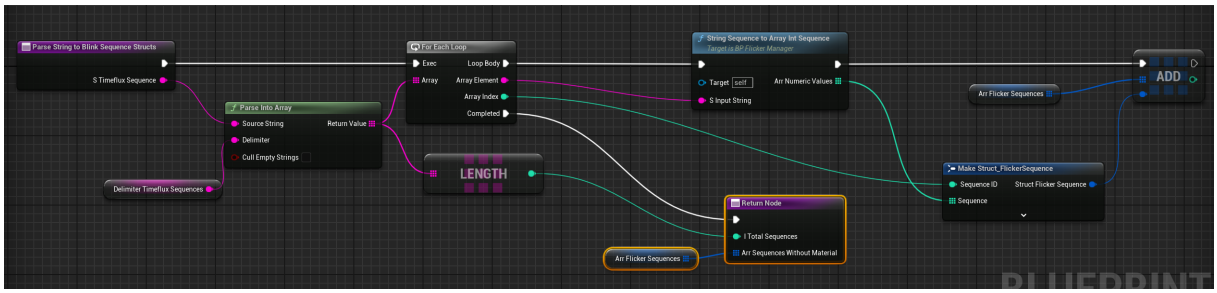

Figure 24: Function ParseStringToFlickerSequenceStruct graph

3.4.11 Function AssignDynamicMaterialToFlickerSequence

As explained earlier, the idea is to create one dynamic material instance per code sequence. This way, each material can be reused every time an object is linked to that particular sequence, without having to create or destroy materials constantly during runtime.

This function:

1. Creates a dynamic material instance from the base material `M_Flicker` for each sequence.

2. Stores each instance inside the corresponding `Struct_FlickerSequence`, in a variable such as `DM_FlickerMaterialInstance`.

3. If the input parameter `PatchesVisible` is set to true, it sets the scalar parameter `Visibility = 1`, so that patches are initially visible.

This function is called after parsing the sequences and before flickering starts, ensuring that every sequence has its own dedicated material prepared and ready for use.

3.4.12 Function UpdateFlickerSequences

This function is the core of the flickering mechanism.

Every time it is called (usually by a timer in the `UpdateBlink` event), it advances the current bit index of all code sequences simultaneously. For example, if each code sequence is 132 bits long, this function will step through bit 0, then 1, 2, 3... until 131, and then cycle back to 0.

For each sequence, it: Extracts the bit corresponding to the current index, and Modifies the `Visibility` parameter of the dynamic material instance linked to that sequence, making the patches appear or disappear.

This flickering effect is the result of toggling visibility over time.

This function returns two outputs: the current bit of the first code sequence and the current index shared by all sequences. The reason for returning both values is that this function must work in both the calibration and gameplay phases.

During calibration, only one object flickers at a time, so it is meaningful to send the actual bit to Timeflux. This allows Timeflux to compare that value directly with the EEG signal. However, during gameplay, multiple code sequences are flickering simultaneously. In this case, it is not meaningful to send a single bit, because each sequence has its own. Instead, only the index of the current bit is sent. Since Timeflux already knows all the sequences, it can reconstruct the correct bit for each one based on the index.

Technically, we could always send just the index—even during calibration—because Timeflux has access to the sequences and could extract the bit from the index. However, to keep the system general and flexible, this function always returns both the bit and the index. The choice of which one to send is made later by `BP_LSL`, depending on the current phase of the experiment. This dual-output strategy introduces no relevant computational expense.

3.4.13 Function GetDistanceToPlayer

This function calculates the 3D distance between a target mesh component and the player's position.

It retrieves the world location of the component and the player, and then computes the distance using the `Distance (Vector)` node. This value is used in perceptual filtering to prioritize closer objects for flickering.

3.4.14 Function FOVFilter

This is the first perceptual filter applied to each mesh component.

It checks whether a component is within the player's Field of View (FOV). **The steps are:**

1. Calculates the direction vector from the player to the component.
2. Normalizes that vector.
3. Computes the dot product with the player's forward vector.
4. Compares the result with the cosine of the FOV angle.

If the result exceeds the threshold, the component is considered within the player's view.

3.4.15 Function LineOfSightFilter

This is the second perceptual filter, applied after the FOV check. It determines whether the component is visible (not fully occluded by other objects), even if it is in the FOV.

Steps:

1. Retrieves the component bounds: origin, box extent, and sphere radius.
2. Offsets trace starting points slightly inward (–5 units) to ensure traces begin from within the object.
3. Dynamically calculates how many traces to use based on object size.
4. Computes evenly spaced directions and key points for the traces.
5. Performs line traces from the player to each point.
6. If any trace reaches the object unobstructed, the function returns `true`.

3.4.16 Function GetMeshComponentsByTag

This function retrieves all mesh components from actors with a specific tag. It then processes those components and organizes them into a structured map based on their tags. The grouping of components depends on the correspondence between actor and component tags. For more on this logic, refer to the tagging rules explained in subsection 3.3.1.

Steps:

1. Retrieves all actors that contain the specified tag.
2. For each actor, it finds the index of that tag in the actor's tag array.

- Note: actors and components can both have multiple tags.
- The key rule is that the tag at index i in the actor must correspond to the tag at index i in the component — they are matched by position.

3. Retrieves all mesh components from the actor.
4. For each component, it checks whether it has a valid tag at the same index:
   - If so, it generates a **formatted tag** using the actor index and the component tag at that position.
   - **Example:** for actor index 1 and component tag 2 → the formatted tag is: `"1_2"`.
   - This formatting is handled by the macro `GetAndFormatComponentTag`.
5. The component is added to the map `Map_TagComponents`, where:
   - The key is the formatted tag (`"1_2"`, `"0_3"`, etc.)
   - The value is a `Struct_PerceptionData`, which will be filled later with visibility and distance information.
6. If none of the components of a given actor have a valid tag at the required index, then all its components are grouped together by default under a single entry.

The resulting groups are stored in the field `StaticMeshComponents` inside their corresponding `Struct_PerceptionData`, and will be processed later by functions like `SetPerceptionData`, `GetChosenComponentsToFlicker`, and `ArrangeIntoCodesStruct`.

3.4.17 Function SetPerceptionData

This function evaluates the perceptual conditions of each component group stored in `Map_TagComponents`.

It checks:

1. Whether the object is inside the player's FOV (FOVFilter)
2. Whether it is visible (not occluded) using LineOfSightFilter
3. Its distance to the player using GetDistanceToPlayer

Optimization: Filters are applied in order. If a component fails the FOV filter, the other two are skipped. This avoids unnecessary calculations.

Distance Initialization: At the start, all components are initialized with a distance value greater than the minimum visible distance. This ensures they will only be selected if later confirmed visible. The user can modify this distance threshold through the variable: `MinVisibleDistance`

Group logic: If components are grouped under the same code sequence , only one of them needs to pass all filters. If at least one component is valid, the entire group will be eligible to flicker.

3.4.18 Function GetChosenComponentsToFlicker

This function selects the most appropriate mesh components to flicker during the current frame.

Steps:

1. Compares the number of available code sequences to the number of filtered component groups.
2. For each available sequence slot:
   - Iterates through `Map_TagComponents`.
   - Selects the closest eligible group (based on distance).
   - Increases its distance to a value beyond `MinVisibleDistance` to avoid re-selection.
   - Adds the group to the array of chosen components.

The output is an array of arrays of mesh components, ready to be assigned to each flicker sequence.

3.4.19 Function ArrangeIntoCodesStruct

This function takes the selected components from `GetChosenComponentsToFlicker` and updates the central array `ArrStruct_FlickerCodeSequences`.

Mechanism:

1. Two sets are defined:
   - One containing the newly chosen components.
   - Another with the components currently linked to each code sequence.
2. For each flicker sequence:
   - If components are still present in both sets → they remain assigned.
   - If they are missing in the new selection of chosen componets → their overlay material is removed.
3. For each new chosen component not yet assigned:

- It is linked to a free flicker sequence.
- The dynamic material instance for that sequence is applied.

4. A list is maintained with the code sequence IDs that changed components. This is later used to trigger the dispatcher `CodesChanged`.

If there are more code sequences than chosen components, some sequences will remain unassigned, and their object list will be empty.

3.5 Communication with Timeflux (BP_LSL)

Timeflux and Unreal Engine communicate using Lab Streaming Layer [21]. See Figure 25.

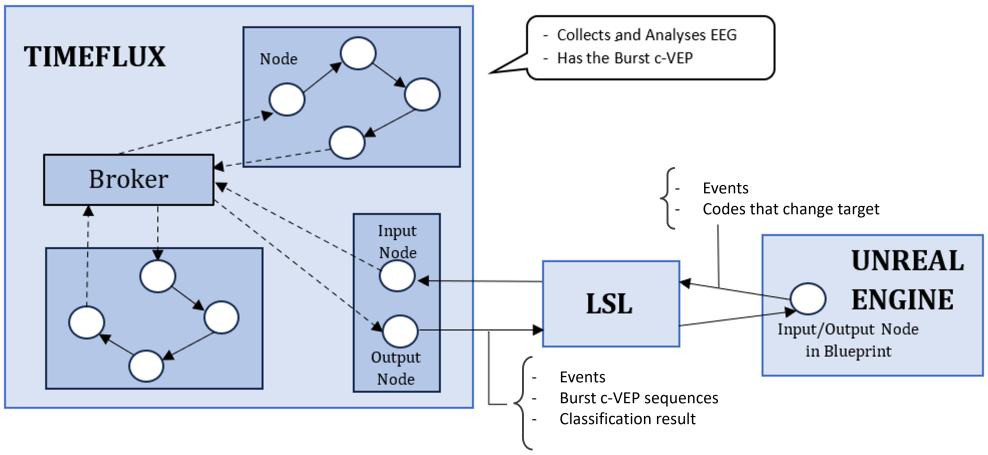

Figure 25: Communication Scheme between Timeflux and Unreal Engine through LSL

3.5.1 LSL Streams and Channels

`BP_LSL` is the actor blueprint responsible for enabling communication between Unreal Engine and Timeflux, using the Lab Streaming Layer (LSL) plugin. It contains three LSL inlet components and two LSL outlet components. See Figure 26.

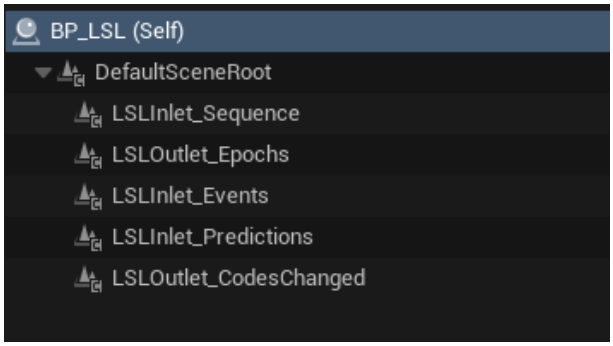

Figure 26: BP_LSL components

LSL Inlets: See Figure 27.

- `LSLInlet_Sequence` — receives the code sequences from Timeflux, both for calibration and gameplay.
- `LSLInlet_Events` — listens for events from Timeflux (stream name: `TimefluxSendsEvents`).
- `LSLInlet_Predictions` — receives predictions from Timeflux (stream name: `TimefluxSendsPreds`).

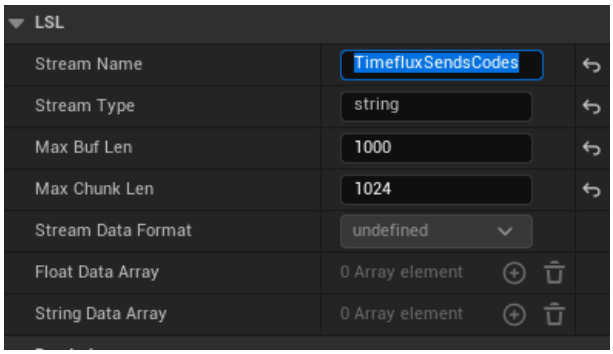

Figure 27: Inlet stream configuration

These three inlets have very similar configurations; the only difference is the name of the stream each one connects to.

LSL Outlets: See Figure 28.

- `LSLOutlet_Epochs` — used to send events such as `epoch`, `calibration_begins`, `session_begins`, `calibration_ends`, `ready`, `run_begins` (see Figure 49 where shows the events send and when). This outlet must be configured with **two channels**, named `label` and `data`. Each channel has:
  - A `Label` (the identifier of the channel, e.g., `label`, `data`)
  - A `Unit` (the actual value to send)

  Example — sending an `epoch`:
  - Channel 0: Label = `label`, Unit = `epoch`
  - Channel 1: Label = `data`, Unit = `0`

  Example — sending `calibration_begins`:
  - Channel 0: Label = `label`, Unit = `calibration_begins`
  - Channel 1: Label = `data`, Unit = `""`

- `LSLOutlet_CodesChanged` — same structure as the Epoch outlet. Used to send the codes whose assigned objects have changed. The stream name is `VRSends_targetChange`.

  Example:

  - Channel 0: Label = `label`, Unit = `codes`
  - Channel 1: Label = `data`, Unit = `1,3,4`

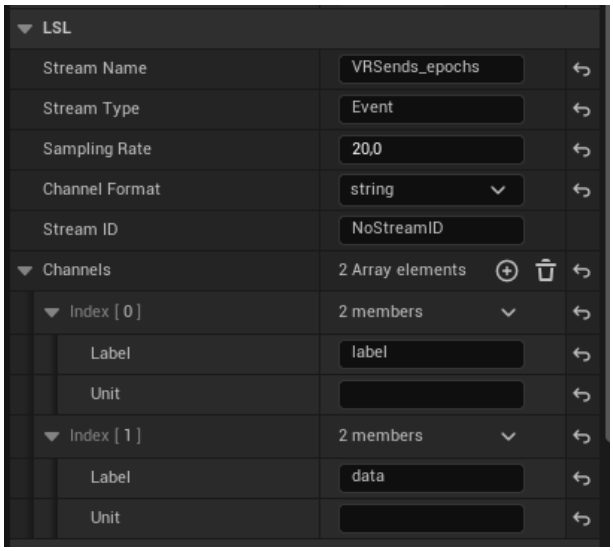

Figure 28: Outlet stream configuration

IMPORTANT:

- The stream names used in Unreal Engine must exactly match those defined in Timeflux.
- If a stream is configured with multiple channels, all of them must be filled in every transmission. Sending a shorter array (e.g., only one value instead of two) will crash Unreal. If you don't want to send anything for one channel, use an empty string (`""`) as a placeholder.

3.5.2 BP_LSL binding with other blueprints

Once the VR session starts, the BP_LSL blueprint immediately sends the session_begins event to Timeflux and binds itself to several events dispatched by other blueprints. These bindings ensure that the LSL communication remains tightly integrated with the core flickering and calibration logic.

IMPORTANT: If any blueprint intends to send information to Timeflux via LSL—and therefore communicate with BP_LSL through event dispatchers—it must be referenced and registered here, and connected to the specific events it needs to trigger within BP_LSL. See Figure 29.

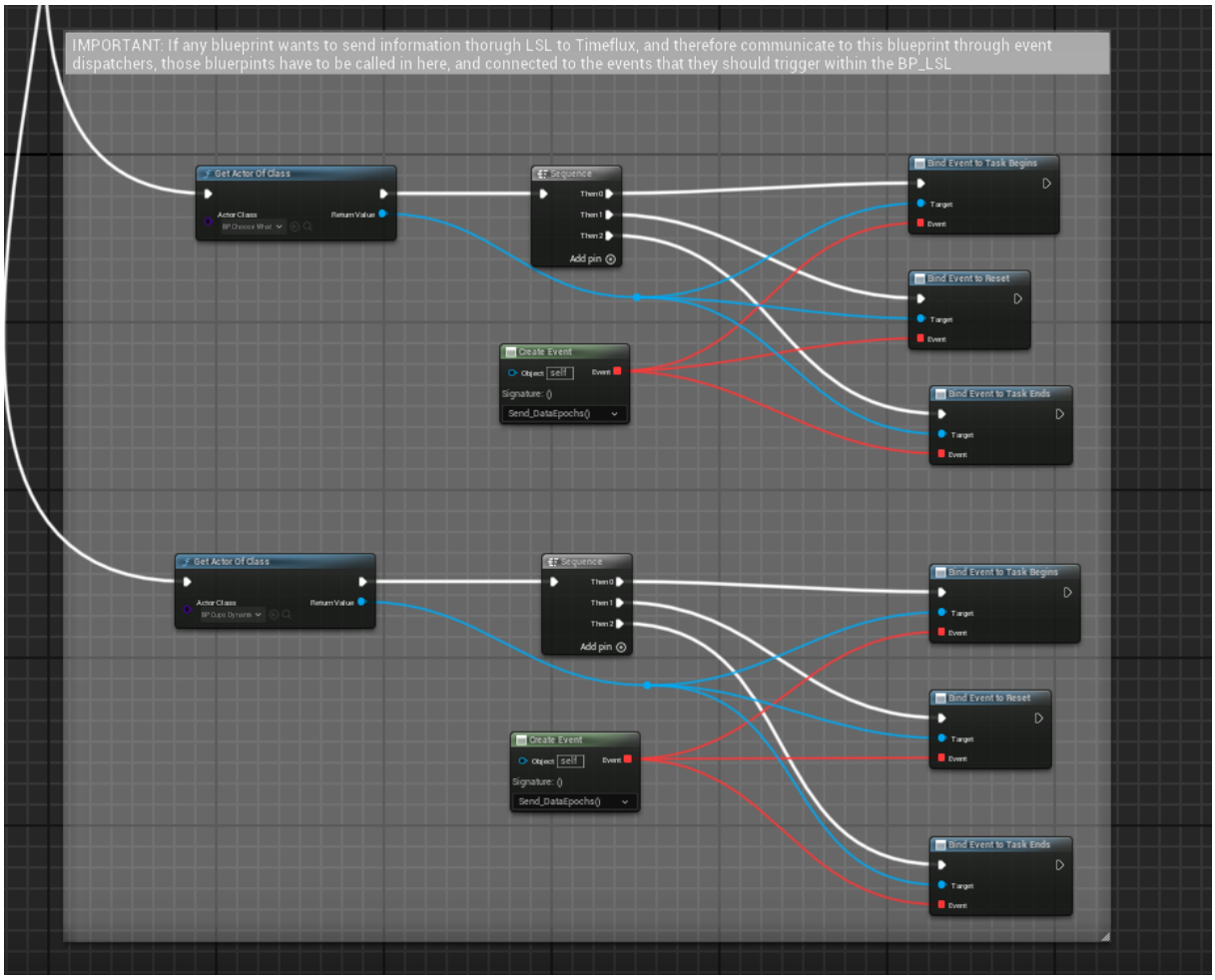

Figure 29: BP_LSL binding with other blueprints

The internal event system in `BP_LSL` consists of a set of event receivers and dispatchers. Each one handles a specific communication task depending on which external blueprint triggers it.

Below, each internal event in `BP_LSL` is described in detail.

3.5.3 Event SendDataEpochs

This event is responsible for sending general status messages to Timeflux. These include: `calibration_begins`, `calibration_ends`, `session_begins`, `ready`, `run_begins`.

It is also used to send the `epoch` events when appropriate (though the bit/index is handled by the specific `EpochAnalysis` event). See Figure 30.

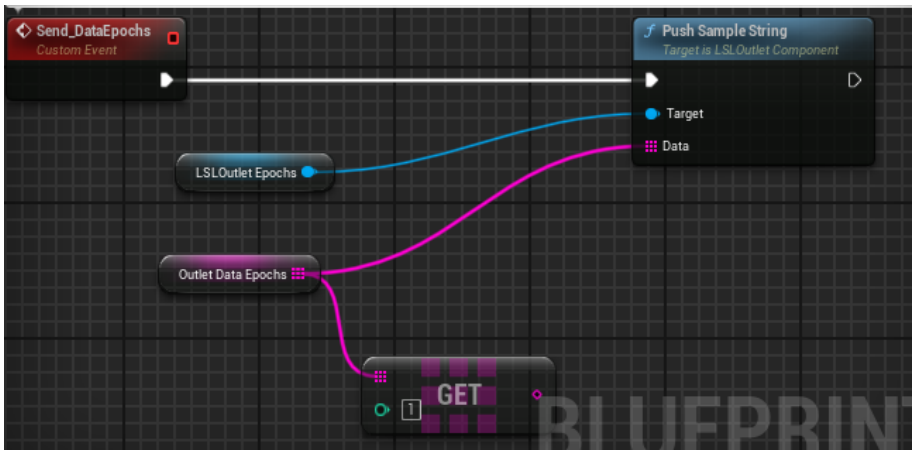

Figure 30: Event SendDataEpochs

Each message is formatted using the LSL outlet with two channels:

- `label` → contains the event name (e.g., `calibration_begins`)
- `data` → usually empty unless it's an `epoch`

This event is used by several blueprints: `BP_Calibration`, `BP_ChooseWhateverCup`, and `BP_CupsDynamic`.

3.5.4 Event EpochAnalysis

This event sends the actual `epoch` event to Timeflux every time the flicker system advances to a new bit.

It sends:

- `label` = `"epoch"`
- `data` = current bit or index (depending on calibration mode)

The choice of sending the bit or the index is based on a boolean that tracks whether the system is in calibration phase. This decision logic is handled within `BP_LSL`, while the bit and index values are received from `BP_FlickerManager`.

Once the decision is made, the event `SendDataEpochs` is called to send this to Timeflux.

3.5.5 Event SendCodesThatChangedTheirObject

This event sends to Timeflux the list of flicker code sequences that have changed their associated mesh components. By doing this, Timeflux can reset the internal momen-

tum tracking for those codes and update its prediction logic to reflect the new visual configuration. See Figure 31.

It uses the `VRSends_targetChange` stream and sends:

- `label` = `"codes"`
- `data` = comma-separated string of changed sequence IDs (e.g., `"1,3,4"`)

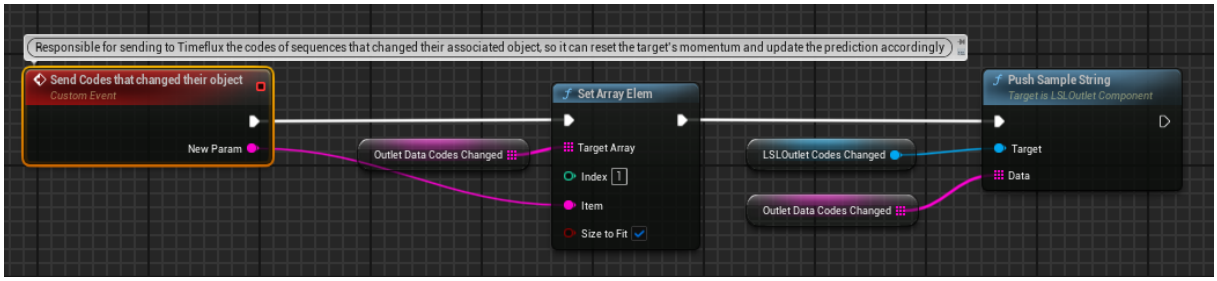

Figure 31: Event SendCodesThatChangedTheirObject

3.5.6 Event OnStreamUpdated_LSLInlet_Sequence

Triggered when new sequence data arrives via the `LSLInlet_Sequence` channel. Since Timeflux may send the same sequences repeatedly, a boolean flag ensures this event only saves them once.

After storing the sequences:

- It triggers `TaskSequencesReceived` and `CalibrationSequencesReceived`, depending on the game state.
- These dispatchers notify other blueprints like `BP_Calibration` or game logic blueprints.

Note: Ideally, Timeflux could send the sequences once, and in the future it could be interesting to see how to send it just once. See Figure 32.

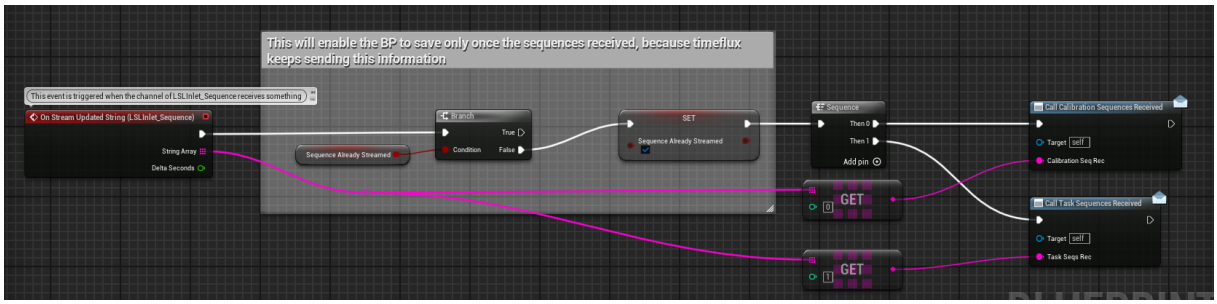

Figure 32: Event OnStreamUpdated_LSLInlet_Sequence

3.5.7 Event OnStreamUpdated_LSLInlet_Events

Triggered when the `LSLInlet_Events` stream receives new data. It listens for a special event:`"ready"`, sent by Timeflux once the model is trained and ready to begin the game.

When this signal is received, it dispatches the event `ReadyEventReceived`, which can be handled by game blueprints such as `BP_ChooseWhateverCup` or `BP_CupsDynamic` to begin gameplay. See Figure 33.

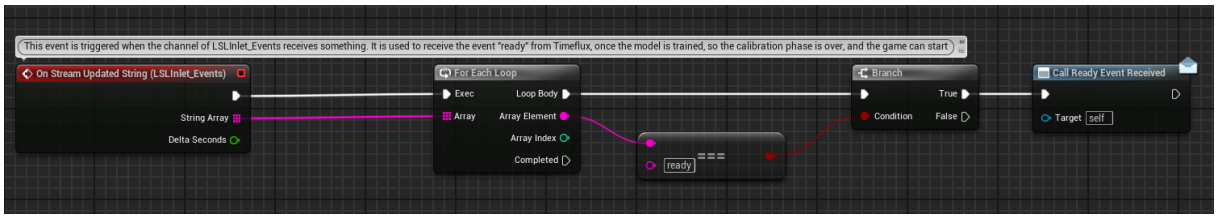

Figure 33: Event OnStreamUpdated_LSLInlet_Events

3.5.8 Event OnStreamUpdated_LSLInlet_Predictions

Triggered when a prediction is received from the `LSLInlet_Predictions` stream.

Currently, Timeflux sends prediction data in JSON format. This event:

- Parses the JSON.
- Extracts only the predicted target (the value of `"target"`).
- Dispatches the `PredictionReceived` event, used by the game logic to act on the prediction.

Note: Ideally, Timeflux should send only the prediction value directly (e.g., an integer) to simplify parsing and reduce risk of errors. See Figure 34.

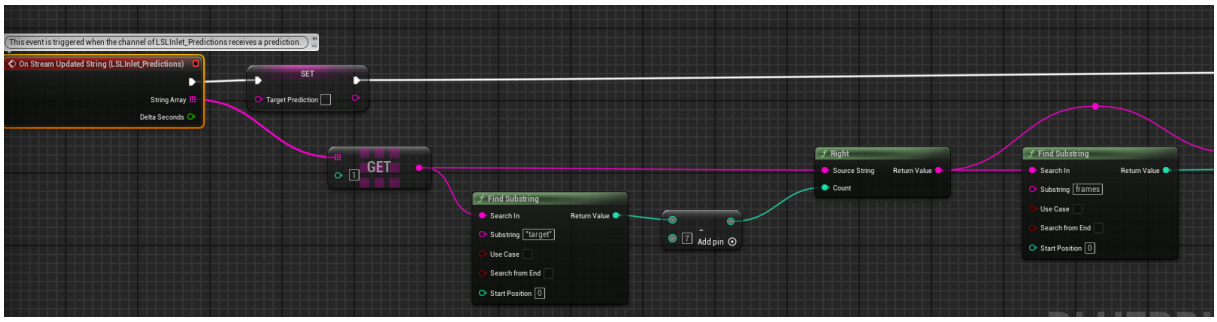

Figure 34: Event OnStreamUpdated_LSLInlet_Predictions

3.6 Calibration Blueprint

`BP_Calibration` is an example actor blueprint that shows how a full calibration routine can be implemented in Unreal Engine. The blueprint handles receiving sequences, structuring the calibration into blocks, managing flickering and pauses, and communicating with Timeflux.

When the game starts, the blueprint retrieves references to `BP_FlickerManager` and `BP_LSL`. It then binds its own `CalibrationBegins` event, which will only be triggered once Timeflux sends the calibration code sequences. No flickering will start until the calibration data is received. See Figure 35.

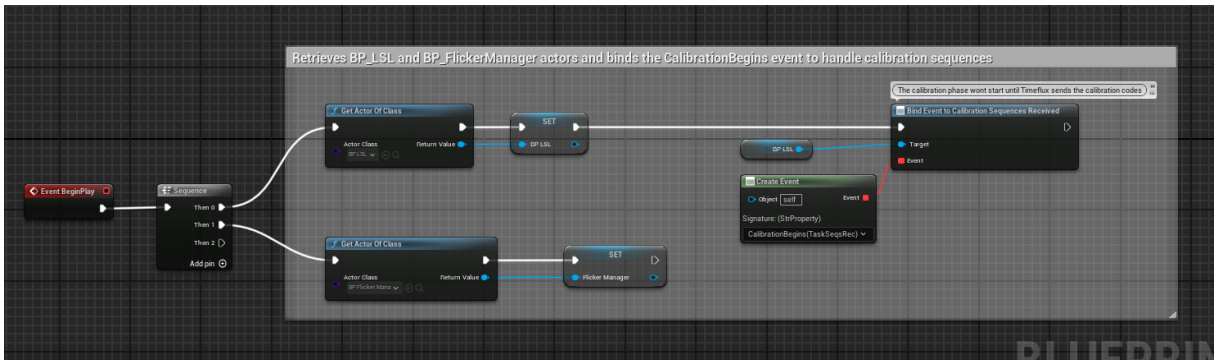

Figure 35: Initialization and Sequence Reception

Once the calibration sequences are received, the `CalibrationBegins` event is triggered. At that point:

- The flickers are set up by calling `SetUpFlickers` with the tag `"calibration"` and the code sequences received from Timeflux. See Figure 36.

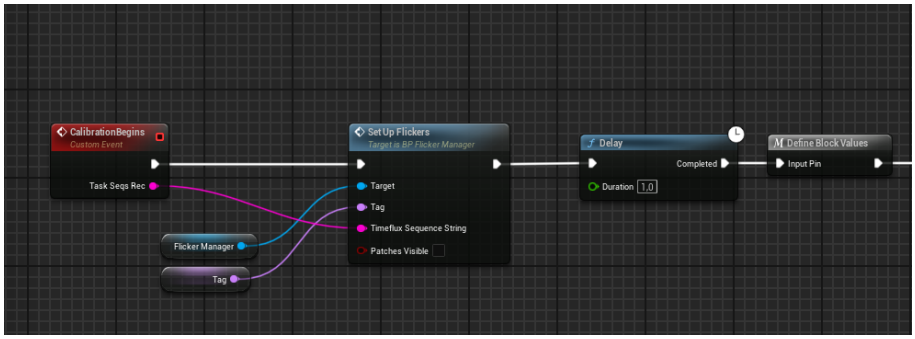

Figure 36: Set up of Flickers with tag "Calibration"

- Timeflux gets noticed of the "calibration_begins". See Figure 37

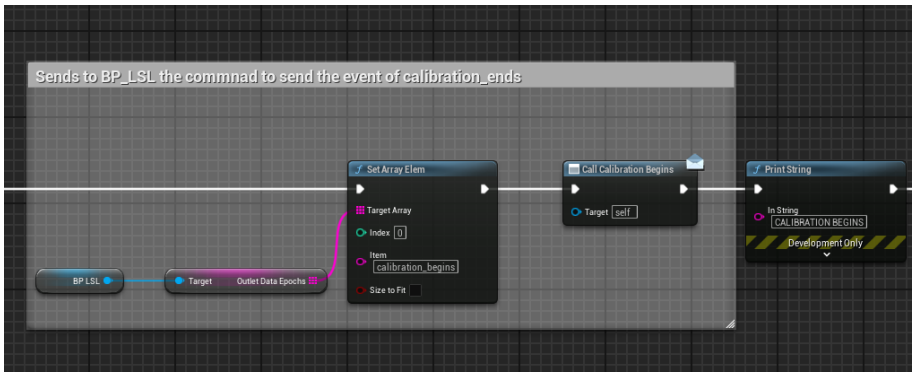

Figure 37: "calibration_begins" is sent to Timeflux

- The timing values for flickering duration and pause between blocks are defined inside the macro `DefineBlockValues`. See Figure 39.

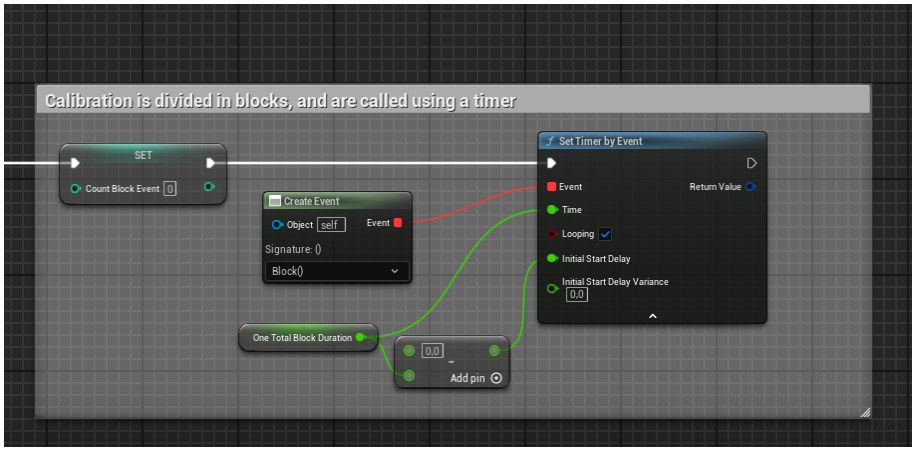

Figure 38: Calibration is separated into blocks

- The timer to do the blocks is called. See Figure 38.

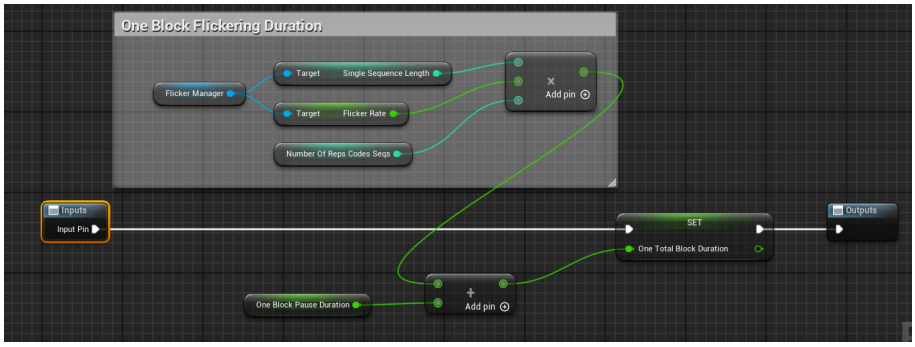

Figure 39: Block values Macro

3.6.1 Structure of Calibration Blocks

The calibration process is divided into alternating flickering and resting blocks. These allow the user to rest between stimulation phases. The progression between blocks is handled by a timer that periodically triggers the event `Block`.

When the `Block` event is called:

- It first checks whether all calibration blocks have been completed. See Figure 40.

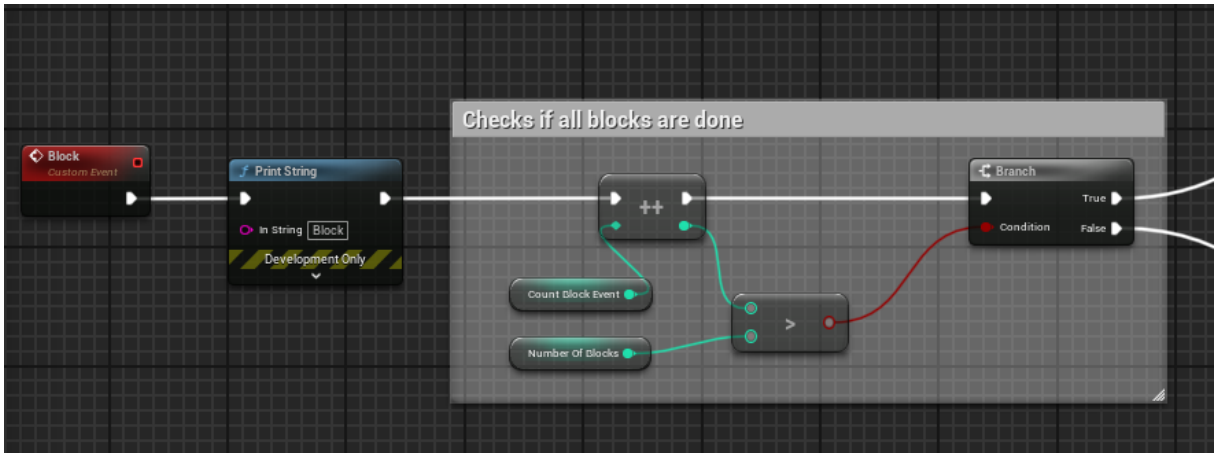

Figure 40: Checking if the calibration blocks are done

- If not:
  - Flickering is paused using `PauseFlickering`, without resetting the code sequence assignments. See Figure 41.

Figure 41: Flickering is paused

  - The bit index is reset to 0 just to ensure synchronization — although it should already be at 0, this is done as a safety check.
  - After the pause period, flickering resumes automatically.

Once all blocks have been executed:

- Flickering is fully stopped by calling `StopFlickering`, resetting both the index and the code sequences, since no more calibration flickers are needed.
- The timer responsible for triggering the `Block` event is cleared.
- A command is sent to `BP_LSL` to emit the `calibration_ends` event to Timeflux. See Figure 42.

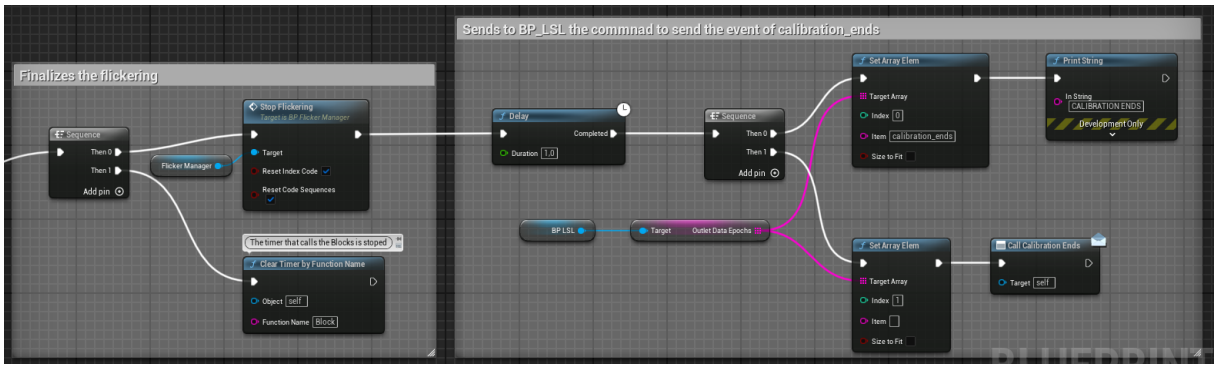

Figure 42: Flickering is stopped and LSL sends "calibration_ends"

3.7 Game Integration Examples

3.7.1 BP_ChooseWhateverCup

The purpose of this blueprint is to provide a minimal interactive demo showcasing how BCI predictions can be mapped to user attention. In this scene, three cube objects are flickering simultaneously, each encoded with a different code sequence. The user is free to look at any of the flickering objects, and when a prediction is received from Timeflux, the corresponding cube is highlighted. This allows the user to verify whether the object they were attending to matches the decoded result.

The system remains active indefinitely and only stops when Unreal Engine is closed, regardless of how many predictions are made.

At the beginning, the blueprint retrieves references to both the `BP_LSL` and `BP_FlickerManager` actors. Once the task sequences and the `Ready` event are received, the game logic starts. See Figure 43.

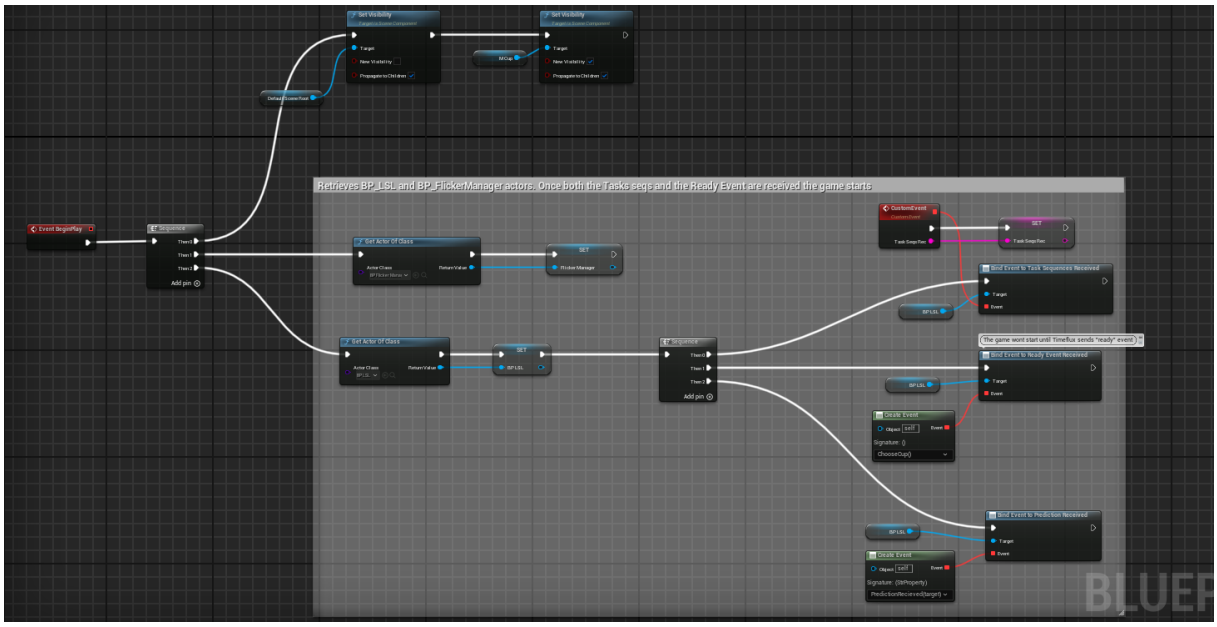

Figure 43: BP_ChooseWhateverCup binding with other blueprints

Upon receiving the `Ready` signal from Timeflux, the event `chooseCup` is triggered. This initiates the run by sending a `run_begins` message to Timeflux and sets up the flickering components with the tag `"Flicker"`, starting the flickering behavior. See Figures 44 and 45.

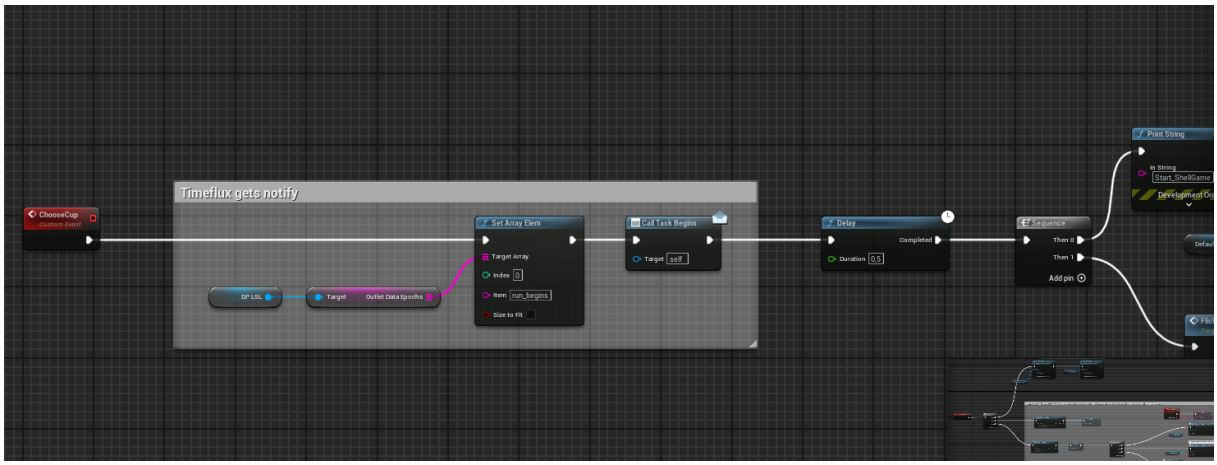

Figure 44: Event ChooseCup part 1

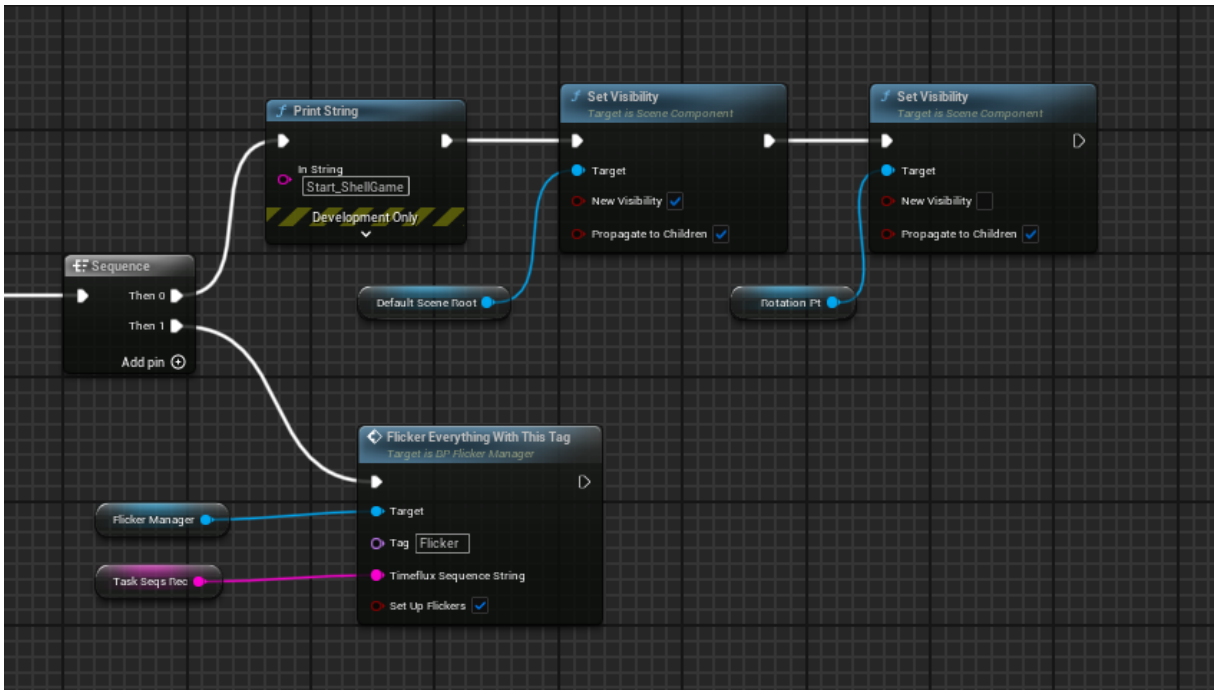

Figure 45: Event ChooseCup part 2

When a prediction is received from Timeflux, the `predictionReceived` event is executed. In this event, flickering is stopped (without resetting the code sequences) to avoid redundant reinitialization. Then, the system looks through the `Flicker Code Sequences` array for the sequence ID that matches the predicted one. If a match is found, the associated mesh components have their overlay material changed to visually highlight the selected object. After a short delay, flickering resumes as before. See Figures 46, 47, and 48.

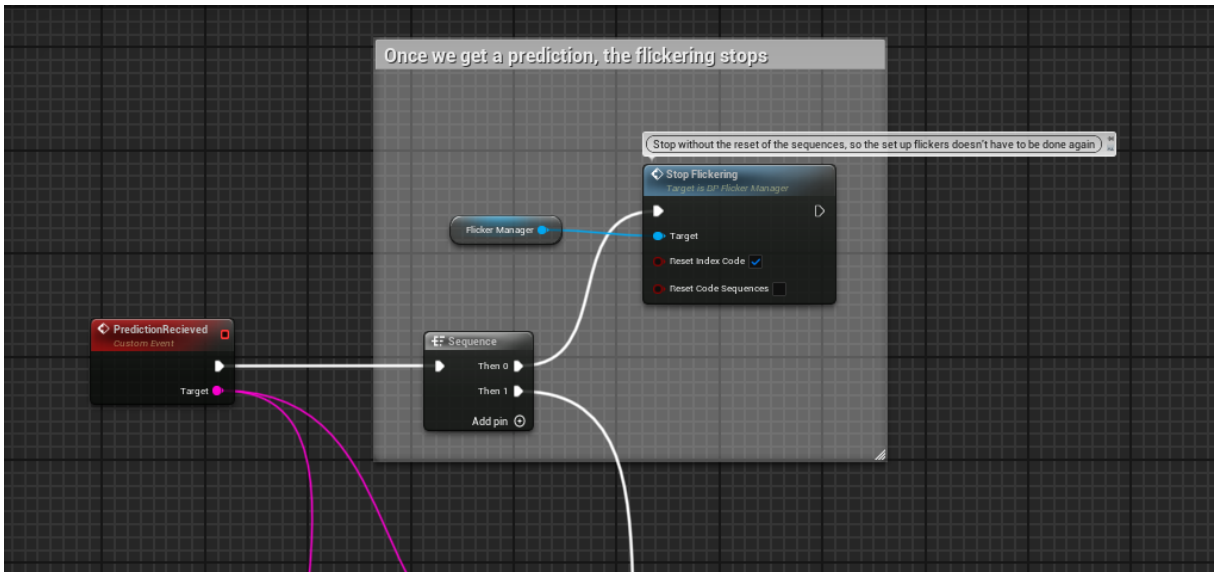

Figure 46: Flickers are stopped once a prediction is received

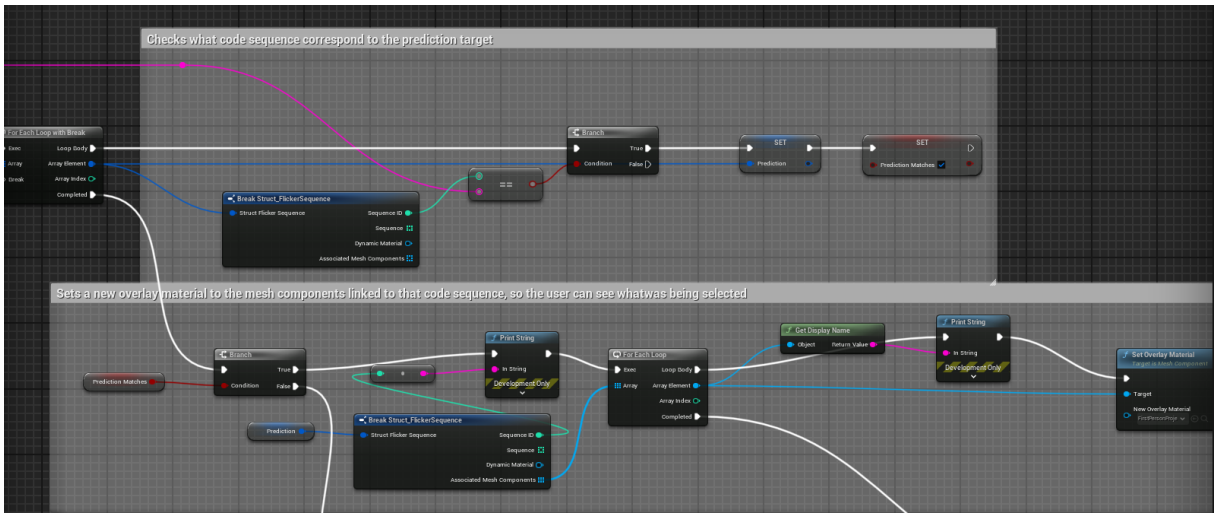

Figure 47: Finds the code that has the same sequence ID as the received prediction

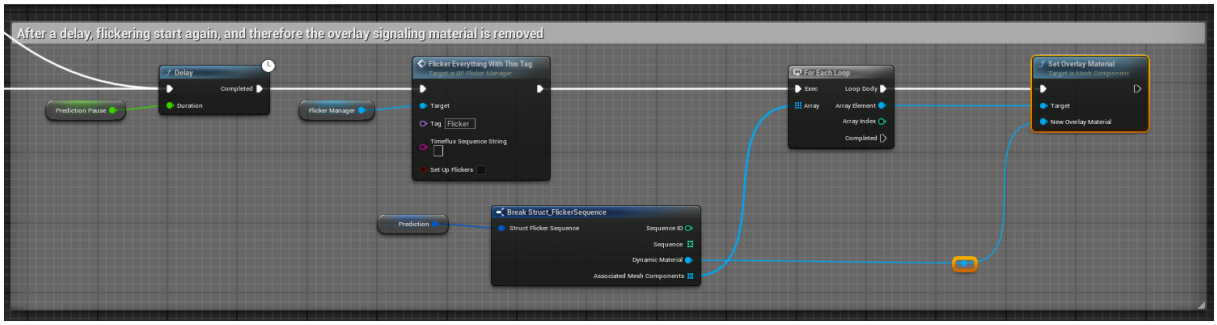

Figure 48: The game restarts

3.7.2 BP_CupsDynamic

This blueprint implements the full shell game logic. Its structure is similar to `BP_ChooseWhateverCup`, but it incorporates additional gameplay elements for visual engagement and user feedback.

Before the cups begin flickering, all three are lifted to reveal the position of a hidden ball. Then, a shuffle sequence takes place, mixing their positions. Once shuffled, the cups start flickering, allowing the user to focus on the one they believe contains the ball. When a prediction is received, the selected cup is highlighted, and all cups are lifted to reveal whether the prediction was correct. After a brief pause, the cups are lowered, and a new shuffle cycle begins, repeating indefinitely.

Two custom events are added in this blueprint:

- **shellGame**: Handles flicker setup, cup elevation/lowering, and the shuffle animation.

- **upDownPrediction**: Manages the logic for raising and lowering the cups in response to a prediction.

The user can customize the behavior of the game via two public variables:

- `VelocityRotation`: Controls the speed of the cup shuffle animation.
- `TotalShuffleRounds`: Determines how many shuffle rotations take place per round.

This blueprint creates a fully interactive and repeatable shell game experience using real-time BCI prediction data.

Phases of the Shell Game, including the Calibration phase, and the events that have to be sent and received between Timeflux and Unreal Engine. See Figure 49.

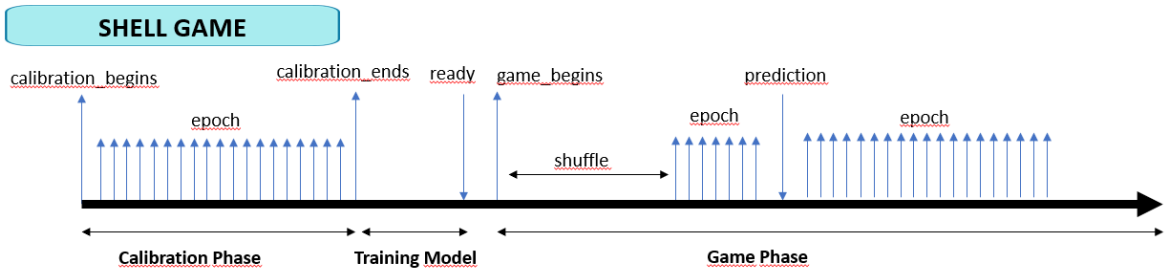

Figure 49: Shell Game events

3.8 User Manual for Editing Variables

This section explains how to edit the key variables of the system, including tags, blueprint parameters, and material values. These can be modified either directly from the main map or from within the respective blueprints and materials.

3.8.1 Editing Variables from the Main Map

There are two main ways to change exposed variables in the system. The first is through the main level map. See Figure 50.

To do this:

1. Open the main level map.
2. Click on the actor or blueprint in the viewport.

3. In the `Details` panel (usually on the right), go to the `Default` category.
4. Here, all editable variables exposed to the map will appear.
5. Modify any values you need, and then click `Save`.

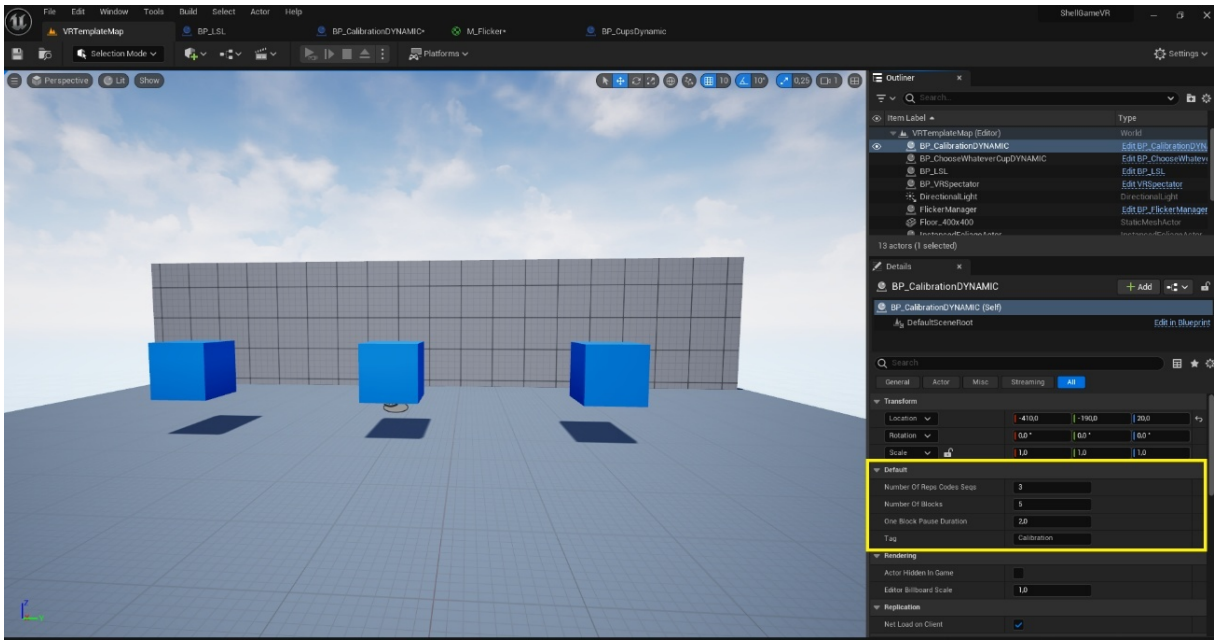

Figure 50: Editing Variables from the Main Map

If you want to modify actor or component tags: See Figures 51 and 52.

- Use the search bar in the `Details` panel and type "`tag`".
- If the actor is selected, you'll see the tags at the actor level.
- If you select a component inside the actor, you'll see the tags specific to that component.

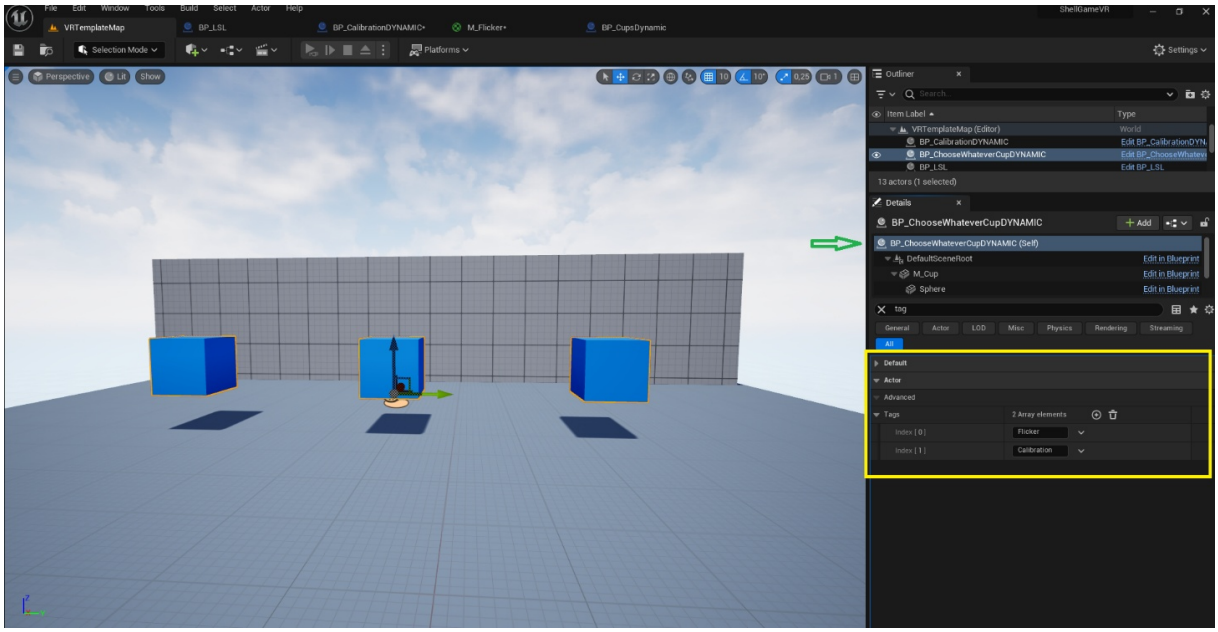

Figure 51: Editing Actor's tags

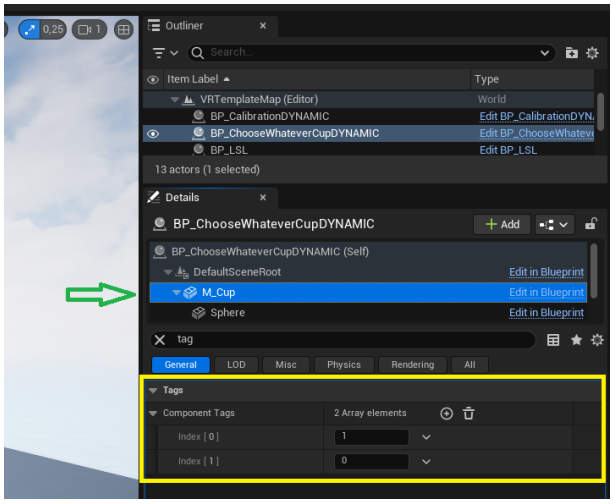

Figure 52: Editing Component's tags

This is the preferred method for adjusting things like tag associations, timing parameters, or any exposed gameplay variable, without having to open the blueprint itself.

3.8.2 Editing Blueprints Internally

The second option is to modify variables or logic directly from inside each blueprint, material, or structure. See Figure 53.

To do this:

- Open the corresponding blueprint (e.g., `BP_FlickerManager`).

- Locate the variable or logic block you want to modify.
- Make the necessary changes and recompile the blueprint.

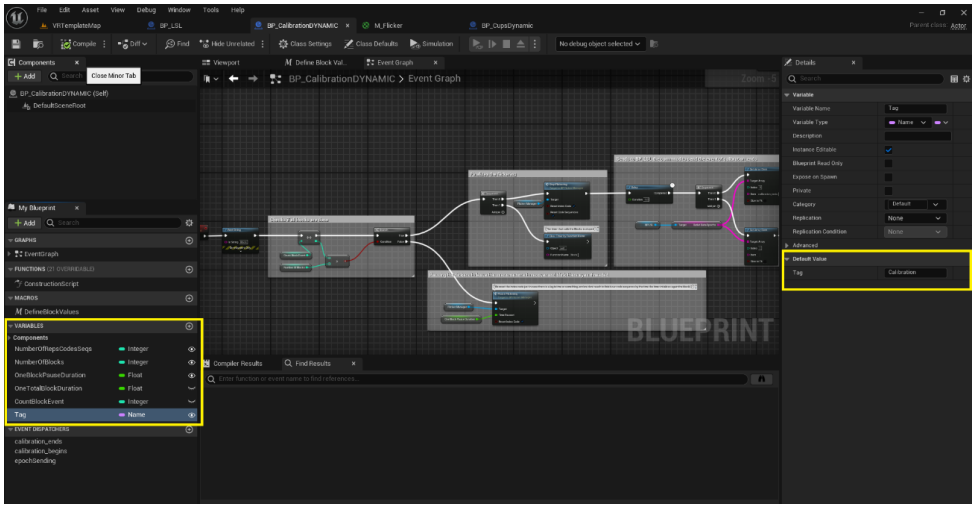

Figure 53: Editing Blueprints Internally

This method is used for adjusting default values that are not exposed to the level, for debugging, or for altering the behavior of functions and events.

3.8.3 Material Parameter Editing

Modifying material parameters requires a slightly different process. In this system, the flickering overlays are handled by dynamic material instances created at runtime.

While it is possible to assign the base dynamic material instance (`DM_FlickerMaterialInstance`) from the map, its internal parameters (like patch size, color, brightness, and visibility) cannot be edited there directly. This is because these parameters are part of the parent material `M_Flicker`, not the instance.

To change them:

1. Navigate to the `Materials` folder inside the imported framework directory. See Figure 54.
2. Locate `M_Flicker`.
3. Open it and edit the exposed parameters inside the material editor.

Alternatively, you can access the material from within `BP_FlickerManager`: See Figure 55.

- Locate the material variable that holds the base material.

- Click the folder icon (green circle) to find its source location.
- Open the parent material from there.

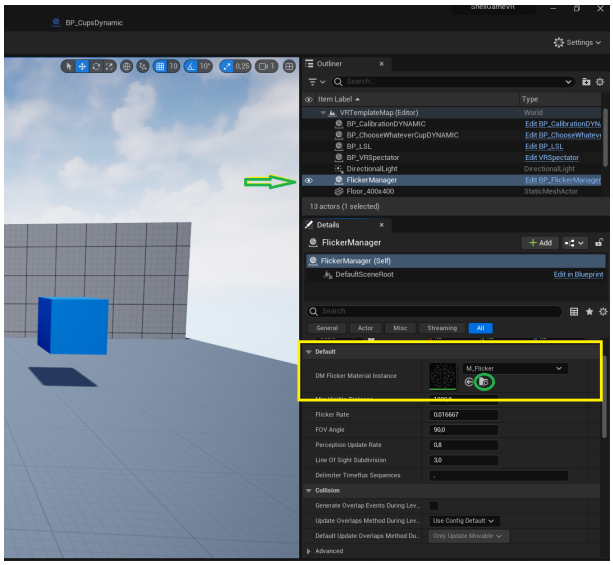

Figure 54: DM_FlickerMaterialInstance inside the `BP_FlickerManager`

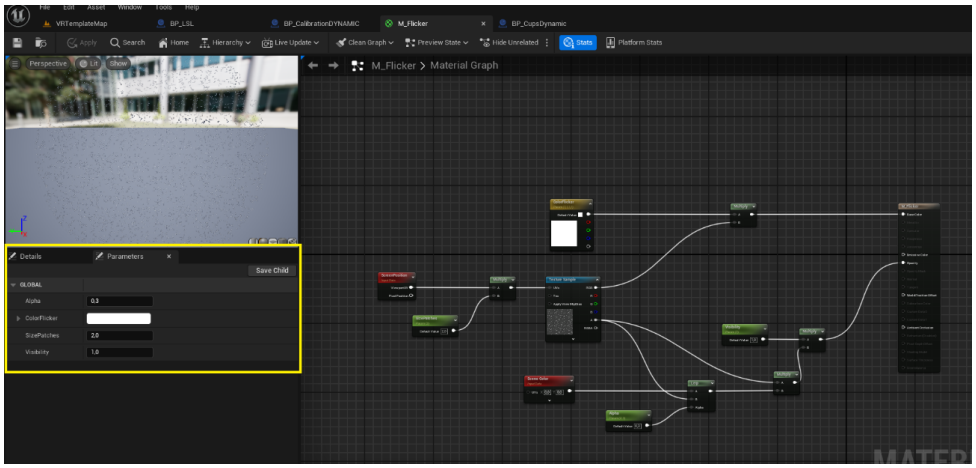

Figure 55: Changing the M_Flicker variables directly inside

Make sure to recompile and save after editing the material so that changes take effect during runtime.

4 Timeflux Framework

Timeflux [22] [7] is an open-source Python framework designed for the real-time acquisition, processing, and classification of time series, with a particular focus on biosignals for BCI

[7] `https://doc.timeflux.io/`

applications. The framework uses a simple YAML-based syntax and follows a modular architecture based on Directed Acyclic Graphs (DAGs), where data and event streams flow directionally between computing nodes. Multiple graphs can run concurrently, each containing an arbitrary number of nodes connected by edges. The original system is composed of six primary graphs:

- **EEG graph**: Handles data acquisition through the Lab Streaming Layer (LSL).
- **Preprocessing graph**: Applies signal conditioning steps, including re-referencing and filtering of EEG signals.
- **Classification graph**: Segments the signal into epochs, applies ERP-enhancement using the Xdawn algorithm, projects the covariance matrix onto its Riemannian tangent space, and classifies the result using a Linear Discriminant Analysis (LDA) model.
- **UI graph**: Manages user interface components for both data monitoring and stimulus delivery.
- **Record graph**: Stores raw and filtered data, as well as all relevant events.
- **Broker graph**: Routes asynchronous messages between graphs using a publish-subscribe communication pattern.

4.1 EEG Preprocessing and Target Prediction

The EEG signal is first preprocessed by applying a bandpass filter between 1 and 25 Hz and re-referencing the signal to the POz electrode. No baseline correction is applied. The continuous signal is then segmented into overlapping windows of 250 ms, with a step size of 2 ms, yielding a classification rate of 500 decisions per second.

Each of these windows is processed using xDAWN spatial filtering (with four components), and the resulting data is projected into the Riemannian tangent space after computing the covariance matrix. These feature vectors are then classified using a Linear Discriminant Analysis (LDA) model to estimate whether a given window contains a burst (event-related potential) or not. The output is a binary value: 1 if a burst is detected, 0 otherwise.

The system accumulates these binary values over time to build an ongoing sequence of neural responses. At a rate of 60 Hz (once per frame), the current sequence is compared to each of the known *burst c-VEP* code sequences associated with the available targets. A Pearson correlation is computed between the observed sequence and each code, and the results are used to update an internal score for each target.

This score acts as a momentum-like value that reflects how strongly the incoming brain data aligns with each target over time. Once the correlation score of a given target crosses a predefined threshold, that target is selected as the prediction.

To make the prediction process both robust and reactive, the system implements an *accumulated momentum* strategy, defined in the `AccumulatedSteadyPred` class. This mechanism gradually builds confidence for a target by counting consistent partial predictions across time, using an exponential function to weigh temporal consistency more heavily.

$$M(x) = 2^{\left(\frac{x}{m}\right)} - 1$$

where:

- x is the number of consecutive partial predictions for a given target,
- m is the number of consecutive predictions required to reach a momentum of 1.

Each time a target receives a new consistent prediction ($x \to x+1$), its momentum is increased by $M(x) - M(x-1)$. At the same time, all other targets have their momentum decreased by the same amount, ensuring that coherent history is preserved but does not dominate indefinitely.

Once the momentum of a target exceeds a predefined threshold, the system emits the corresponding object prediction. At this point, the comparison process is interrupted, and the system proceeds with the next step in the interaction flow. This strategy allows the classifier to react quickly while still maintaining robustness in the presence of noise or small prediction inconsistencies.

4.2 Changes done in Timeflux

4.2.1 General Configuration and `mainVR.yaml` Structure

To implement Timeflux in conjunction with VR and Unreal Engine, several modifications to the original setup were required. One of the main changes was the creation of the `mainVR.yaml` configuration file (see Appendix A), which is responsible for connecting the various graphs and nodes needed for real-time processing.

This file imports multiple subgraphs corresponding to each processing stage:

- Device acquisition (`DEVICE.yaml`)
- Preprocessing (`preprocessing.yaml`)
- Classification (`classification_md_plot.yaml`)
- ERP computation (`erp.yaml`)
- Optional data recording (`record.yaml`)

It also defines two main graphs:

- **Broker graph:** Contains a central ZeroMQ node (`timeflux.nodes.zmq.Broker`) that handles asynchronous messaging across graphs.
- **VR graph:** Manages the bi-directional communication between Timeflux and Unreal Engine. It includes:
  - `DataframeMerge` node, which formats the calibration and task sequences.
  - `Sub` and `Pub` nodes for ZMQ-based communication.
  - LSL inlets and outlets to send and receive data such as predictions, events, and incoming EEG epochs.
  - Custom utility nodes to convert incoming string data into dataframes.
  - A `debug` node used to monitor the flow of information.

It is important to note that the names of all streams must match exactly those defined in Unreal Engine to ensure successful data transmission.

4.2.2 Communication with Unreal Engine via LSL and ZMQ

Three LSL outlets are used to send information from Timeflux to Unreal:

- `TimefluxSendsCodes` for code sequences,
- `TimefluxSendsPreds` for predictions,
- `TimefluxSendsEvents` for session_begins, calibration_begins, calibration_ends, ready, run_begins.

Two LSL inlets are used to receive information from Unreal:

- `VRSends_epochs` receives EEG epochs,
- `VRSends_targetChange` receives target change events.

The communication is managed through a `sub` node subscribing to the topics `predictions`, `events`, and `filtered`, which are then forwarded to their corresponding LSL outlets.

4.2.3 Custom Data Formatting Nodes

In order to structure and process the data correctly within Timeflux, custom Python nodes were developed:

- `DataframeMerge`: sends a DataFrame with calibration and task codes to Unreal.
- `StringToDictionary_epochs`: parses and converts incoming EEG epoch strings into structured JSON.

index	label	data
timestamp	"someLabel"	{"index": 34, "bits": [0]}

- `StringToDictionary_targetChange`: processes target change strings.

These nodes ensure compatibility with Timeflux's data requirements and enable correct downstream processing.

This is done in the file `data_utils` (see Appendix B).

4.2.4 LSL Timestamp Synchronization Fix

A major issue encountered during development was the mismatch between the internal Timeflux clock and LSL timestamps. This misalignment caused incorrect EEG-event associations and therefore inaccurate predictions.

To solve this, the original timestamp handling logic was replaced with a new implementation that:

- Anchors all incoming data to Timeflux's internal clock,
- Computes the relative time offsets of LSL timestamps,
- Reconstructs the adjusted timestamps accordingly.

Old implementation:

```
if self._offset != 0:
    stamps = np.array(stamps) + self._offset
stamps = pd.to_datetime(stamps, format=None, unit="s")
self.o.set(values, stamps, self._labels, self._meta)
```

New implementation:

```
base_time = np.datetime64(now(), "us")
relative_offsets = np.array(stamps) - stamps[0]
relative_offsets = (relative_offsets * 1e6).astype("timedelta64[us]")
adjusted_stamps = base_time + relative_offsets
adjusted_stamps = pd.to_datetime(adjusted_stamps)
self.o.set(values, adjusted_stamps, self._labels, self._meta)
```

This solution significantly improved time alignment and classification accuracy.

4.2.5 Shape Node Fix in Classification Graph

Another necessary fix was in the classification nodes defined in `classification.yaml` and `classification_md_plot.yaml`. These graphs used the `Shape` transformer from `timeflux.estimators.transformers.shape`, which produced an error due to variable naming issues (notably with an extra underscore). A new file shapeAux inside the nodes.VR folder was created, in order to not modify the base file from Timeflux.

4.2.6 Several Predictions Handling

In the `accumulation_base.py` module, the node `AbstractAccumulation` was modified to support multiple predictions and avoid discarding valid signals. The following line was added inside the `update(self)` method, directly after the comment `# ignore stale signals`:

```
self._recovered = True
```

4.2.7 Target Reset Handling in Prediction Logic

A new function was implemented in `md.py` to properly reset the internal momentum tracking for those codes and update its prediction logic to reflect the new visual configuration.This is done everytime a code sequence has changed their associated mesh components, maintaining consistency in the prediction logic and avoiding false positives driven by previously accumulated context.

```
def targetChange(self, target):
    self._consec[target] = 0
    self._momentum[target] = 0
```

To invoke this reset behavior during runtime, the following condition was added inside the `update(self)` method of the `AbstractAccumulation` class:

```
if self.i_targetChange.data is not None:
    target = self.i_targetChange.data["data"]
    self.targetChange(target)
```

To support the `targetChange` mechanism described above, both `classification_md_plot.yaml` and `classification.yaml` configuration files were modified to include an additional input channel. This new channel receives external target change events and routes them to the `AbstractAccumulation` node via the `i_targetChange` inlet.

4.2.8 Event Publication for Classification Synchronization

The system was updated to emit classification events in addition to predictions. Events such as `ready` now allow full synchronization with Unreal Engine. Unreal engine wont start the shell game until it receives this event and makes sure that the training of the model was successfully done.

5 Future Work

Although the current system is fully functional, there are several areas identified for future development and refinement. These improvements aim to enhance system performance, improve user interaction, and increase experimental reliability:

- **Integrating eye-tracking:** to evaluate user attention and perception in real time, and to enable gaze-based filtering of flickering stimuli. This hybrid approach leverages the complementary strengths of neural and ocular signals to enable faster, more reliable, and more natural interaction. Ultimately, it supports the broader ambition to push BCIs out of the lab and into real-world applications that are intuitive, accessible, and user-centered, facilitating the development of applications across various domains such as clinical diagnostics, aerospace control systems, and hands free interaction interfaces.
- **Optimizing Blueprints:** by reducing computational load in frequently executed operations, especially those involved in perception filtering and flicker logic.
- **Migrating key logic to C++:** to improve performance, enable finer memory control, and reduce overhead from Blueprint interpretation.
- **Revisiting the HUD-based flicker method:** which was initially discarded due to occlusion and performance limitations, but could be re-implemented under improved engine conditions or with optimized code.
- **Running controlled experiments:** to evaluate the perceptual and signal quality effects of variables such as patch size, brightness, and color. This would help identify optimal parameter values based on user performance and EEG response.

- **Improving classification strategies:** by exploring alternative prediction methods, including hybrid models that combine BCI signals with gaze or behavioral data.
- **Reducing the dependency on constant epoch transmission:** Instead of sending an `epoch` event every frame, the system could be restructured to rely on the known frequency and start time of the stream, reducing data traffic and simplifying synchronization.
- **Standardizing data sent to Timeflux:** For example, sending only the sequence index (rather than the actual bit) during both calibration and gameplay, as Timeflux already stores the sequence information and can extract the relevant bit internally.

6 Conclusion

This project successfully demonstrated the feasibility and potential of implementing a reactive Brain-Computer Interface (BCI) system based on burst code-modulated Visual Evoked Potentials (c-VEP) within a fully immersive Virtual Reality (VR) environment. Through the development of a modular and scalable framework using Unreal Engine, Timeflux, and dry EEG technology, real-time brain-controlled interaction was enabled using visually evoked neural responses alone.

Key contributions of this work include:

- The design and integration of textured visual stimuli (Ricker patches), optimized for foveal stimulation and minimal peripheral distraction, significantly improving user comfort and classification accuracy.
- The implementation of a dynamic material overlay system in Unreal Engine to apply flickers directly onto 3D objects, enabling real-time control over multiple independent flicker sequences.
- The development of a perception-based filtering mechanism to dynamically allocate flickers only to visible objects, maintaining signal integrity and system performance in VR contexts.
- A robust communication pipeline using Lab Streaming Layer (LSL) to synchronize EEG data, visual stimuli, and prediction feedback across all system components.

Experimental validation using a VR shell game confirmed the system's ability to detect user intent with high accuracy and low latency, marking a significant step toward practical, real-world BCI applications beyond the lab.

Future work will focus on integrating eye-tracking technology, optimizing blueprint logic, and conducting user-centered evaluations to further enhance system performance and usability.

References

[1] Gianluca Di Flumeri, Pietro Aricò, Giuseppe Borghini, Nicola Sciaraffa, Andrea Di Florio, and Fabio Babiloni. The dry revolution: Evaluation of three different eeg dry electrode types in terms of signal spectral features, mental states classification and usability. *Sensors*, 19(6):1365, 2019.

[2] Simon Ladouce et al. Portable dry-electrode eeg systems for real-world bci: Limitations and solutions. *Journal not specified*, 2022. Full citation details not available from provided documents; placeholder entry.

[3] Lukáš Vařeka and Simon Ladouce. Prediction of navigational decisions in the real-world: A visual p300 event-related potentials brain-computer interface. *International Journal of Human–Computer Interaction*, 37(12):1152–1163, 2021.

[4] D. Zhu, J. Bieger, G. Garcia Molina, and R. M. Aarts. A survey of stimulation methods used in ssvep-based bcis. *Computational Intelligence and Neuroscience*, 2010:702357, 2010.

[5] Te Cao, Feng Wan, Piotr U. Mak, Donny Mak, Mang I. Vai, and Yong Hu. Objective evaluation of fatigue by eeg spectral analysis in steady-state visual evoked potential-based brain-computer interfaces. *BioMed Research International*, 2014:1–10, 2014.

[6] Frédéric Dehais, Hasan Ayaz, Aleksandra Kawala-Sterniuk, Simon Ladouce, et al. Non-invasive brain computer interfaces: Challenges and opportunities. In *Neuroergonomics and Cognitive Engineering*. Springer, 2024. Chapter discussing practical limitations of rBCIs including long SSVEP calibration times.

[7] Felix Gembler, Martin Spüler, and Tobias Mönch. Dynamic time window mechanism for time synchronous vep-based bcis. *PLOS ONE*, 13(6):e0218177, 2018.

[8] Jordy Thielen, Pim van den Broek, Jason Farquhar, and Peter Desain. Broad-band visually evoked potentials: Re(con)volution in brain-computer interfacing. *PLOS ONE*, 10(7):e0133797, 2015.

[9] Víctor Martínez-Cagigal, Jordy Thielen, Elena Santamaría-Vázquez, Sara Pérez-Velasco, and Roberto Hornero. Brain–computer interfaces based on code-modulated visual evoked potentials (c-vep): A literature review. *Journal of Neural Engineering*, 18(6):061002, 2021.

[10] Kalou Cabrera Castillos, Simon Ladouce, Ludovic Darmet, and Frédéric Dehais. Burst c-vep based bci: Optimizing stimulus design for enhanced classification with minimal calibration data and improved user experience. *NeuroImage*, 284:120446, 2023.

[11] William J. Harrison and Peter J. Bex. A unifying model of orientation crowding in peripheral vision. *Current Biology*, 25(24):3213–3219, 2015.

[12] Jost B. Jonas, Uwe Schneider, and Gottfried O. H. Naumann. Count and density of human retinal photoreceptors. *Graefe's Archive for Clinical and Experimental Ophthalmology*, 230(6):505–510, 1992.

[13] Thomas Euler, Silke Haverkamp, Timm Schubert, and Tom Baden. Retinal bipolar cells: elementary building blocks of vision. *Nature Reviews Neuroscience*, 15(8):507–519, 2014.

[14] John A. Greenwood, Bilge Sayim, and Patrick Cavanagh. Crowding is reduced by onset transients in the target object (but not in the flankers). *Journal of Vision*, 9(1):2, 2009.

[15] Nicholas J. Priebe. Mechanisms of orientation selectivity in the primary visual cortex. *Annual Review of Vision Science*, 2:85–107, 2016.

[16] Tomomi Ichinose and Samah Habib. On and off signaling pathways in the retina and the visual system. *Frontiers in Ophthalmology*, 2:989002, 2022.

[17] Pasquale Arpaia, Eleonora De Benedetto, Nicola Donato, Luigi Duraccio, and Nicola Moccaldi. A wearable ssvep bci for ar-based, real-time monitoring applications. In *2021 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*, pages 1–6. IEEE, 2021.

[18] Ina Käthner, Andrea Kübler, and Sebastian Halder. Rapid p300 brain-computer interface communication with a head-mounted display. *Frontiers in Neuroscience*, 9:207, 2015.

[19] Byung-Hwan Cho, Jong-Min Lee, Jin-Ho Ku, Dong-Pil Jang, Jae Sung Kim, In-Young Kim, Jae-Hun Lee, and Seong-Il Kim. Attention enhancement system using virtual reality and eeg biofeedback. In *Proceedings IEEE Virtual Reality*, pages 156–163. IEEE, 2002.

[20] Hybrid Black. Unicorn hybrid black unity interface. `https://github.com/unicorn-bi/Unicorn-Hybrid-Black-Unity-Interface`, 2024. Accessed: 2025-03-26.

[21] LSL-Website. LSL-Website. `https://labstreaminglayer.org/`. Accessed: 2024.

[22] Pierre Clisson, Romain Bertrand-Lalo, Marco Congedo, Guillaume Victor-Thomas, and Jean Chatel-Goldman. Timeflux: An open-source framework for the acquisition and near real-time processing of signal streams. In *Proceedings of the 8th International Brain-Computer Interface Conference*, 2019.

[23] EPICA. Howto add player to user interface #UE4 #rendertarget 2020 edition. YouTube, January 2020.

[24] Mathew Wadstein Tutorials. What is? HUD - draw texture nodes in unreal engine 4 (UE4). YouTube, February 2017.

[25] Nils Gallist. How to cast in unreal engine 5. YouTube, April 2022.

[26] Unreal game dev with Yaz. UE - Get 3D character into 2D widget. YouTube, January 2023.

A mainVR.yaml

```
import:
  # Device graph : sensor
  - graphs/{{DEVICE}}.yaml
  # Preprocessing graph
  - graphs/preprocessing.yaml
  # Classification graph
  - graphs/classification_md_plot.yaml
  # graphs/classification.yaml
  # ERP computation graph
  - graphs/erp.yaml
  # Record graph
  {% if RECORD_DATA == 'true' %}
  - graphs/record.yaml
  {% endif %}
graphs:

  # Broker node
  - id: Broker
    nodes:
    - id: proxy
      module: timeflux.nodes.zmq
      class: Broker

  # VR Graph
  - id: VR
    nodes:
    - id: dataframesMerge
      module: nodes.VR.data_utils
      class: DataframeMerge
      params:
        df1: {% for CODE in CALIBRATION_CODES.split() %}
                - "{{CODE}}"
              {% endfor %}
        df2: {% for CODE in TASK_CODES.split() %}
                - "{{CODE}}"
              {% endfor %}


    - id: subVR
      module: timeflux.nodes.zmq
      class: Sub
      params:
        topics: [filtered, predictions , events]


    - id: lsl_outlet_codes
      module: timeflux.nodes.lsl
      class: Send
      params:
        name: TimefluxSendsCodes
        type: Event
        format: string
        rate: 1.0
        source: my_source
```

```
      config_path: null

  - id: lsl_outlet_preds
    module: timeflux.nodes.lsl
    class: Send
    params:
      name: TimefluxSendsPreds
      type: Event
      format: string
      rate: 1.0
      source: my_source
      config_path: null

  - id: lsl_outlet_events
    module: timeflux.nodes.lsl
    class: Send
    params:
      name: TimefluxSendsEvents
      type: Event
      format: string
      rate: 1.0
      source: my_source
      config_path: null

  - id: lsl_inlet_epochs
    module: timeflux.nodes.lsl
    class: Receive
    params:
      prop: name
      value: VRSends_epochs
      timeout: 1.0
      channels: null
      max_samples: 1024
      clocksync: true
      dejitter: false
      monotonize: true
      threadsafe: true
      config_path: null

  - id: lsl_inlet_targetChange
    module: timeflux.nodes.lsl
    class: Receive
    params:
      prop: name
      value: VRSends_targetChange
      timeout: 1.0
      channels: null
      max_samples: 1024
      clocksync: true
      dejitter: false
      monotonize: true
      threadsafe: true
      config_path: null

  - id: modifyData_epochs
    module: nodes.VR.data_utils
    class: StringToDictionary_epochs
```

```
    - id: pubVR_epochs
      module: timeflux.nodes.zmq
      class: Pub
      params:
        topic: events

    - id: modifyData_targetChange
      module: nodes.VR.data_utils
      class: StringToDictionary_targetChange

    - id: pubVR_targetChange
      module: timeflux.nodes.zmq
      class: Pub
      params:
        topic: targetChange

    - id: debug
      module: nodes.VR.debug
      class: DebugNode


    edges:
      - source: dataframesMerge
        target: lsl_outlet_codes
      - source: subVR:predictions
        target: lsl_outlet_preds
      - source: subVR:events
        target: lsl_outlet_events
      - source: lsl_inlet_epochs
        target: modifyData_epochs
      - source: modifyData_epochs
        target: pubVR_epochs
      - source: lsl_inlet_targetChange
        target: modifyData_targetChange
      - source: modifyData_targetChange
        target: pubVR_targetChange
    rate: 20
```

B data_utils.py

```
import importlib
import pandas as pd
import json
from timeflux.core.node import Node
from timeflux.helpers.clock import now

class DataframeMerge(Node):
    def __init__(self, df1, df2):
        self.df1 = df1
        self.df2 = df2
        self.axis = 1
```

```
        self.ignore_index = False  # Ignorar índices
        self.round = 0

    def update(self):
        self.o.data =pd.DataFrame({
                "calibration_codes":[str(self.df1)],
                "task_codes":[str(self.df2)]})
        if self.round == 0:
            self.round+=1

class StringToDictionary_epochs(Node):
    def __init__(self):
        self.strArray = None

    def update(self):
        if self.i.ready():
            index_value = self.i.data.index[0]
            i_label = str(self.i.data["label"].iloc[0])
            i_data = str(self.i.data["data"].iloc[0]).replace(" ", "")

            params = i_data.split(";")

            o_data = {}
            for param in params:
                if ":" in param:
                    key, value = param.split(":")
                    if key == "bits":
                        o_data[key] = [int(x) for x in value.strip('[]').split(',')]
                    else:
                        o_data[key] = int(value)
                else:
                    o_data[param] = ""

            o_data_json = json.dumps(o_data)

            self.o.data = pd.DataFrame([[i_label, o_data_json]],
                columns=["label", "data"], index=[index_value])
            self.o.meta = self.i.meta


class StringToDictionary_targetChange(Node):
    def __init__(self):
        self.strArray = None

    def update(self):
        if self.i.ready():
            index_value = self.i.data.index[0]
            i_label = str(self.i.data["label"].iloc[0])
            i_data = str(self.i.data["data"].iloc[0]).replace(" ", "")

            self.o.data = pd.DataFrame([[i_label, i_data]],
                columns=["label", "data"], index=[index_value])
            self.o.meta = self.i.meta
```

C HUD Method

This method aims to implement the Heads-Up Display (HUD) across the main display. The objective is to project 3D objects into 2D space and overlay the flicker image onto these projections. To achieve the 2D projection, a component called "Scene Capture Component 2D" was utilized. This component functions as a camera that records any object, simulating a perspective from the character's viewpoint, which is typical in first-person games. This component was integrated into the first-person character blueprint. [23], [24], [25], [26]

After positioning the camera at the character's head position, its settings were configured as follows:

- A render texture target was created to save the recorded information.
- The composite mode was set to overwrite; otherwise, when the capture is taken on each tick, it will overlap instead of refreshing.
- The primitive render mode was set to "Use ShowOnlyList." This setting is fundamental as it specifies which objects to record. When set to this mode, only the objects specified in this list are recorded.
- The capture source was configured to record with SceneColor in RGB and inverse opacity in the alpha channel.

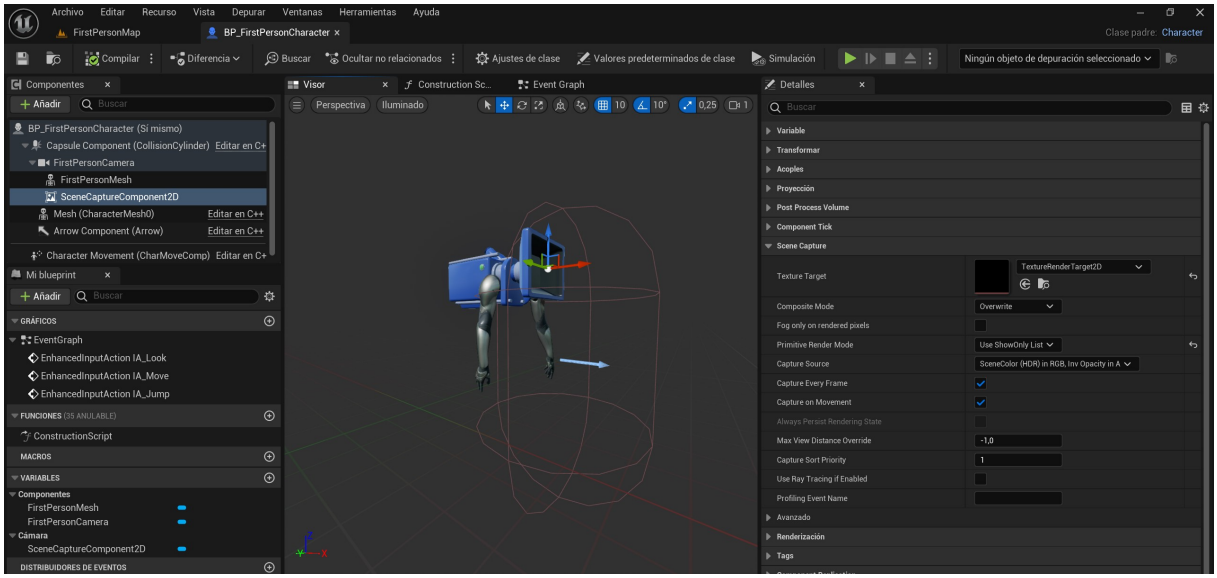

Figure 56: Scene Capture Component 2D in the first-person character blueprint

The next step involved integrating this render target into a HUD widget that spans the entire scene. To achieve this, a material utilizing the render target was created.

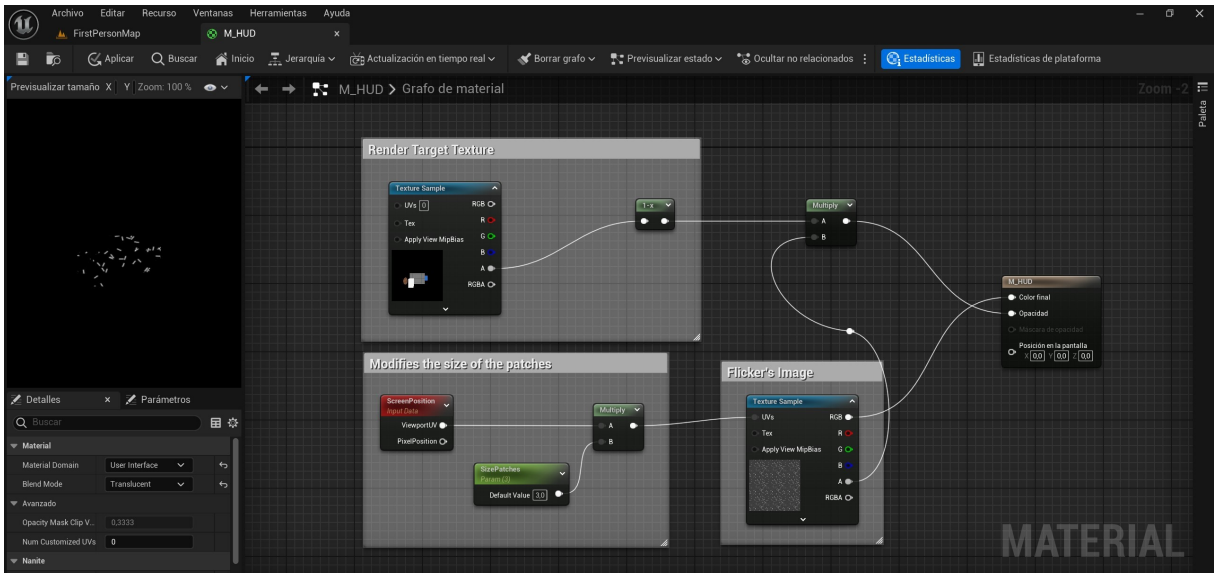

Figure 57: Material setup for HUD with flickers

The overview of the material configuration is as follows. It is necessary to establish the material domain as User Interface, as this will be the material used in the widget. Additionally, the Blend Mode must be set to translucent to allow visibility of the background.

The material creation process involves three main components:

- **The render target texture:** Only the alpha channel is needed to obtain the shape of the objects. Since the SceneCaptureComponent2D was set to capture with inverse opacity in the alpha channel, it is necessary to invert it back.
- **The texture sample containing the image with the patches:** To display the color of the patches, the RGB node of this texture is connected to the base color of the material. Additionally, to utilize the shape of the patches, this alpha channel is multiplied by the one obtained in the render target texture sample. Thus, the result after the multiply node is a shape of the objects in the render target, with the interior filled with the patch particles. This result is connected to the opacity input of the material node.
- **Configuration of the size of the patches:** The remaining nodes are used to configure the size of the patches as desired. To apply these operations to the flicker texture sample, simply add to the UV's input (the texture coordinates).

Afterwards, a HUD widget needed to be created and configured to match the screen size. This ensures that, when overlaid, the patches appear on top of the objects. The configuration of this widget involved three key steps:

1. Initializing the Widget: First, it was necessary to specify that upon starting the game (using the "begin play event"), the widget should be created and displayed on

the screen. This was implemented in the game level blueprint, as this blueprint is always used when the game is initialized.

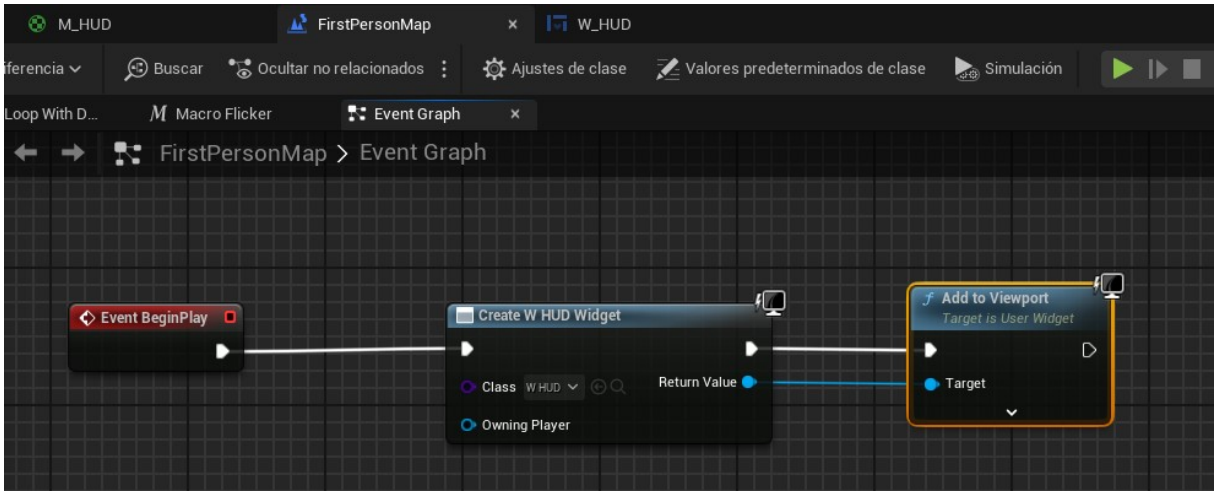

Figure 58: Widget initialization in the Level Blueprint "FirstPersonMap"

2. Configuring the Widget Image: As previously mentioned, the widget includes an image box that spans the entire screen. The position is set by configuring the coordinates in the "Anclas" section. Additionally, in the "Apariencia" section, the image source is set to the material created earlier. The tint of this image can also be configured, allowing for modification of the patch colors (e.g., green, red, white, etc.). In Figure 59 it was chosen green. The opacity of the patches can also be adjusted using the alpha channel of the tint parameter (see Figure 60).

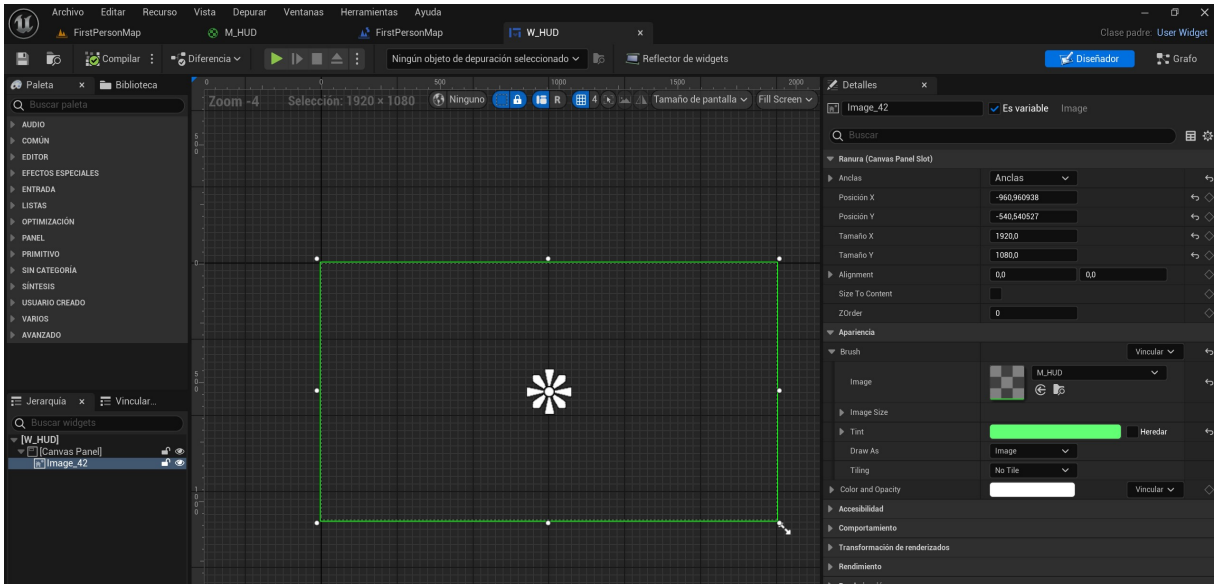

Figure 59: Widget image configuration

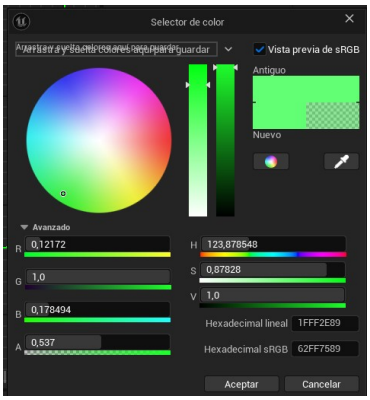

Figure 60: Color choice for the tint

3. Configuring Events within the Widget: This section involves setting up the flicker effects, ensuring that the patches blink according to the desired sequence, with each object following a different sequence.

In Figure 61, the configuration of the HUD widget blueprint is shown. When the widget is created (Event Construct node), the ShowOnlyList, which contains the objects recorded by the SceneCaptureComponent2D, is first obtained. Subsequently, all objects in the game tagged with a specific label —in this case, the chosen label is "Flicker"— are collected and added to the ShowOnlyList. Following this, the "Macro FlickerHUD" is called (a macro is used to encapsulate and reuse a set of Blueprint nodes). In this case, it is called for each object in the list, with only the index needing to be changed.

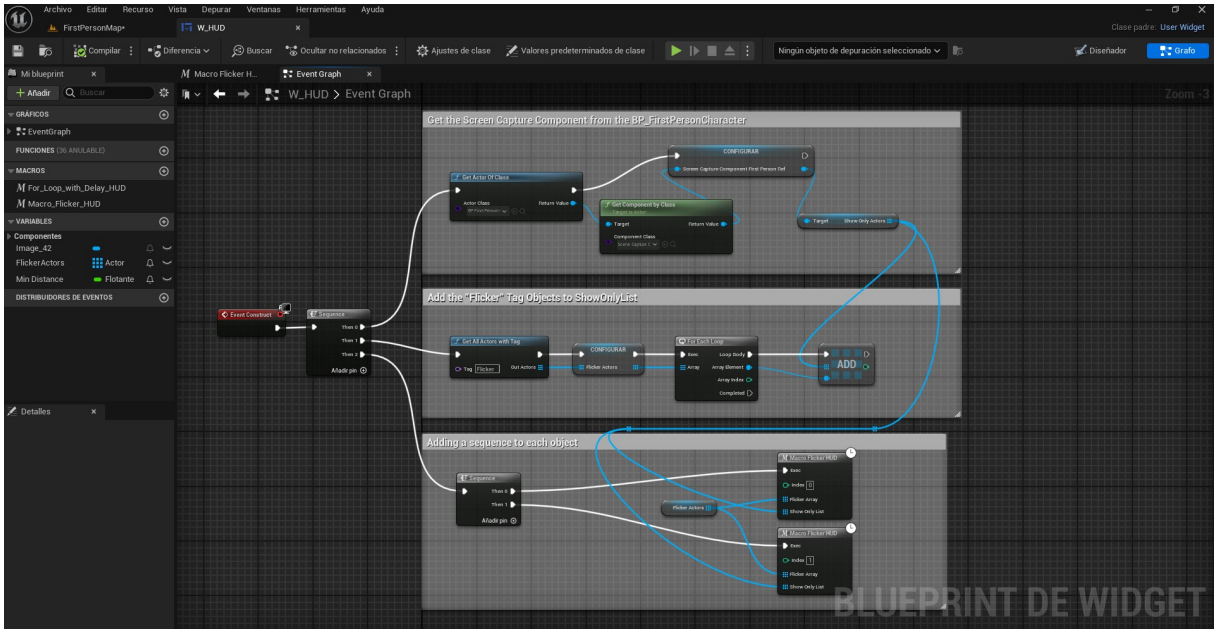

Figure 61: Configuration of the HUD widget blueprint

In Figure 62, the blueprint for the "Macro Flicker HUD" is presented. This macro is responsible for separating the different sequences represented in an array of strings. Using

the index provided as an input to the macro, it identifies the sequence corresponding to each object.

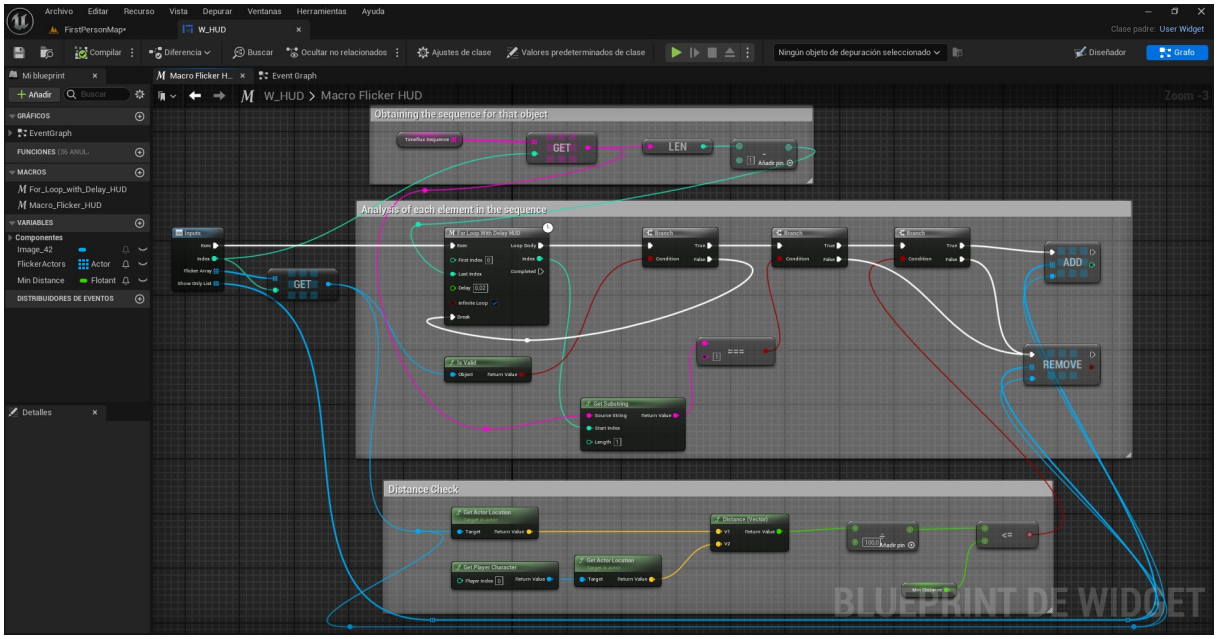

Figure 62: Blueprint of the Macro "Macro Flicker HUD"

Additionally, this macro utilizes a custom "For Loop with Delay" . This loop instructs the program to iterate through each number in the sequence, taking a delay between each number. This delay allows for the modification of the flicker speed. It can also be specified whether the loop should be infinite or not. Within this loop, two checks are performed: the distance of the character from the object, so that if the character moves too far away, the object stops flickering, and whether the current sequence number is 1 or 0, to determine the visibility of the patches. If the number is 1, the patches are visible and the object is included in the ShowOnlyList; if the number is 0, the object is excluded from the list. This enables each flicker to follow independent sequences.