**Installation**

**MonkeyLogic Installation:**

Since we are running a single task for any Unity experiment, the installation is quite simple and requires no modification from these files.

* Get latest zip archive version from: <https://monkeylogic.nimh.nih.gov/download.html>
* Extract archive to installation directory.
* Open command prompt and type:
  + cd MonkeyLogic/task/
  + git clone --recurse-submodules https://github.com/Doug1983/MTLab\_ML\_UnityTask.git
* Download the latest liblsl for Matlab release from: <https://github.com/labstreaminglayer/liblsl-Matlab/releases>.
* Extract the contents of the Liblsl for Matlab release file to: *MonkeyLogic/task/MTLab\_ML\_UnityTask/libLSL/bin/*
* If the installation does not work (i.e. liblsl does not load in Matlab) you can run this script: *MonkeyLogic/task/MTLab\_ML\_UnityTask/libLSL/build\_mex.m*

**Unity Installation**

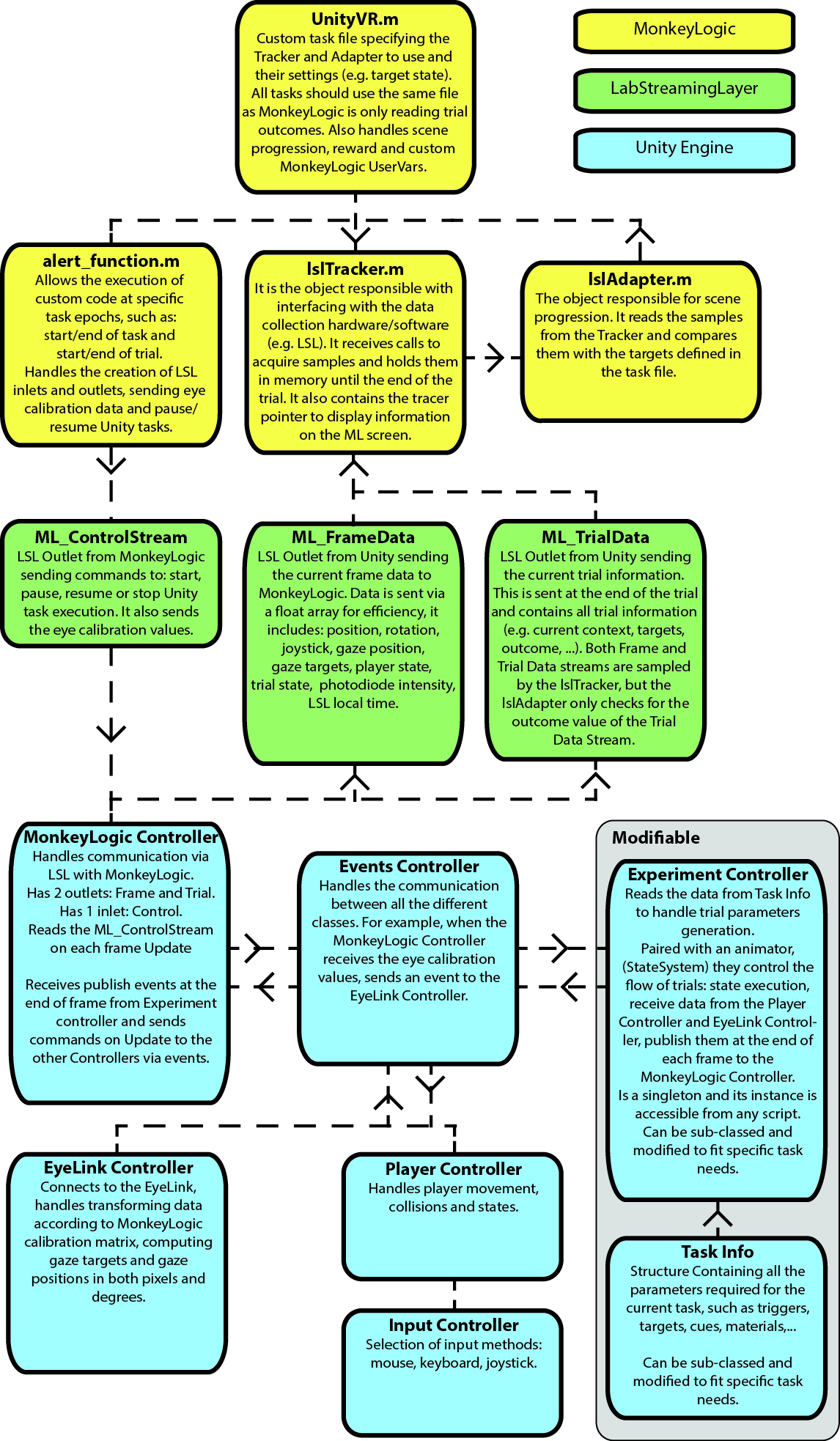
This installs everything needed for the Unity example tasks to work, including the LabStreamingLayer for Unity scripts.

* Get Unity Hub and install version *2019.3.0b3*.
* Create a base directory for the Unity project.
* From the command line, navigate to the directory and run:
  + git clone --recurse-submodules https://github.com/Doug1983/MTLab\_UnitySource.git .

**EyeLink SDK**

The SDK is necessary to have Unity access the EyeLink and retrieve eye samples and can be downloaded here: <https://drive.google.com/open?id=1ggGMG3ZsGim3Runcfe7JXZoaC2rzDwap>

It also uses a specific DLL file that is provided by SR-Research without the source code. It is included in the UnitySource repository and should be in: *UnityProject/Assets/Scripts/EyeLink/DLLs*/*interop.SREYELINKLib.dll*

**Architecture**

**MonkeyLogic**

In MonkeyLogic, each task is defined by a series of *scenes*. For example, in a delayed match to sample task, a single trial would be composed of 4 consecutive scenes:

* Fixation: fixation point appears and waits for a certain fixation duration
* Cue: target/cue appears for a specific duration
* Delay: maintain fixation for a certain delay
* Response: fixation point disappears, target and distractor appear and waits for response to either locations

Each individual scene is a combination of inputs (i.e. *Trackers*) and possible nested conditions (i.e. *Adapters*). Here are some examples:

* Fixation *Scene*: the eye data is sampled (*EyeTracker*) to check if (condition = *Adapter*) it falls within the fixation window around the fixation point (*SingleTarget*), then checks if (condition = *Adapter*) fixation lasts for a certain time (*WaitThenHold*) before terminating the *Scene*.

*WaitThenHold***(** *SingleTarget***(** *EyeTracker* **) )**

* Response *Scene*: the eye data is sampled (*EyeTracker*)to check if it falls within the fixation window of either the target or distractor within a predefined time limit (*MultiTarget*).

*MultiTarget***(** *EyeTracker* **)**

For UnityVR tasks, there is only a single *Scene*: *lslAdapter***(** *lslTracker* **)**. Where the *lslTracker* samples the ML\_TrialData stream waiting for a trial outcome value (e.g. Correct, Incorrect, EarlyResponse, …) to terminate the trial properly, administer reward and save the ML\_FrameData to file.

**LabStreamingLayer**

In LSL, input/output streams are defined by multiple parameters:

* Name: stream name to be discoverable over the network.
* Content type: kind of data sent over the network, such as EEG, Markers (i.e. strings), Unity, …
* Number of channels: data will be a channel x sample array
* Rate: Irregular (0) or any value in Hz
* Channel format: cf\_string, cf\_float32, cf\_double64, cf\_int32, …
* Unique ID: unique identifier from the source/device

The next table shows the 3 streams used by the system. More details on Frame and Trial data streams will be provided in the Unity section.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Direction** | **Name** | **Content Type** | **Channel #** | **Rate** | **Channel Format** | **Unique ID** |
| ML -> Unity | ML\_ControlStream | Markers | 1 | 0 | cf\_string | control1214 |
| Unity -> ML | ML\_FrameData | Unity | 22 | 0 | cf\_double64 | frame1214 |
| ML\_TrialData | Markers | 1 | 0 | cf\_string | trial1214 |

ML\_FrameData

Has 22 “channels” coming from Unity, to which 4 are added in MonkeyLogic, defined here:

1. Player position X
2. Player position Y
3. Player position Z
4. Player rotation
5. Joystick position X
6. Joystick position Y
7. Player collision state: instance ID of collider object such as triggers or targets
8. Gaze position X: position in degrees
9. Gaze position Y
10. Gaze target ID #1: instance ID of foveated object
11. Gaze target ID #2
12. Gaze target ID #3
13. Gaze target ID #4
14. Gaze target ID #5
15. Gaze ray counts #1: number of rays hitting gaze target #1 out of 33
16. Gaze ray counts #2
17. Gaze ray counts #3
18. Gaze ray counts #4
19. Gaze ray counts #5
20. Trial state index: integer value of trial state
21. Photodiode intensity: grayscale value of flashing square to synch monitor with photodiode
22. Unity local LSL time
23. LSL time correction between Unity and ML clocks: remote clock = local clock + time correction
24. Sample time: timestamp in Unity clock time of sample being pushed to stream
25. ML local LSL time: timestamp when sample is obtained
26. ML trial time: time in milliseconds of ML trial (all other ML events use this time scale)

ML\_TrialData

Sent out over the network as a single JSON formatted string, containing information on:

* Trial number
* Start position
* Fixation objects, materials and positions
* Cue objects, materials and positions
* Target objects, materials and positions
* Distractor objects, materials and positions
* Outcome

**Unity**

OOP primer

Unity uses (for now) Object Oriented Programming at its core and some concepts are important to understand the current implementation. We’ll create a fake game to illustrate them.

Let’s assume we want to first create the *base* for all possible living things that can be found in the game. To define a set of properties (i.e. variables) and methods (i.e. functions) that the things can have to interact with one another, we create a *Class*:

public abstract class LivingThing : MonoBehaviour

{

public string Name;

// This is a comment. Default values can be defined here.

// If not set, the properties’ values will be null.

protected float health = 100;

private float weight;

}

First, a LivingThing *inherits* the properties and methods of the *base* class MonoBehaviour (the basic Unity class for all Game Objects). However, it’s ill-defined, meaning that it’s still missing a lot of key properties (e.g. Is it human or animal? How many legs? What’s its speed?). Another word of ill-defined class is abstract, meaning that an *object* of this class cannot be created or *instantiated*, without first having a *sub-class* or another class *inheriting* from it.

public class Human : LivingThing

{

private string job = “farmer”;

public bool IsMale = true;

protected virtual void Start()

}

Name = “Billy-Bob”;

health = 50;

weight = 255; // This will return an error.

}

We have defined our Human sub-class that is inheriting the basic properties of the LivingThing class, added a few more properties and defined a Start() function. The Start function is executed once for each object when it is initialized/instantiated. In this case it sets the values of the LivingThing properties, but:

* Name can be changed because its *scope* is public, which is accessible from anywhere
* health can be changed because it’s protected, which is accessible from the base-class and any sub-classes
* weight cannot be changed because it’s private, which is only accessible within the class in which it is defined

The Start function does not return a value (void) and is defined as virtual which means that its sub-classes can override it. For example, if Billy-Bob is actually a werewolf:

public class Werewolf : Human

{

private Color furColor;

private override void Start()

{

base.Start(); // calls the Human Start function

furColor = Color.white;

}

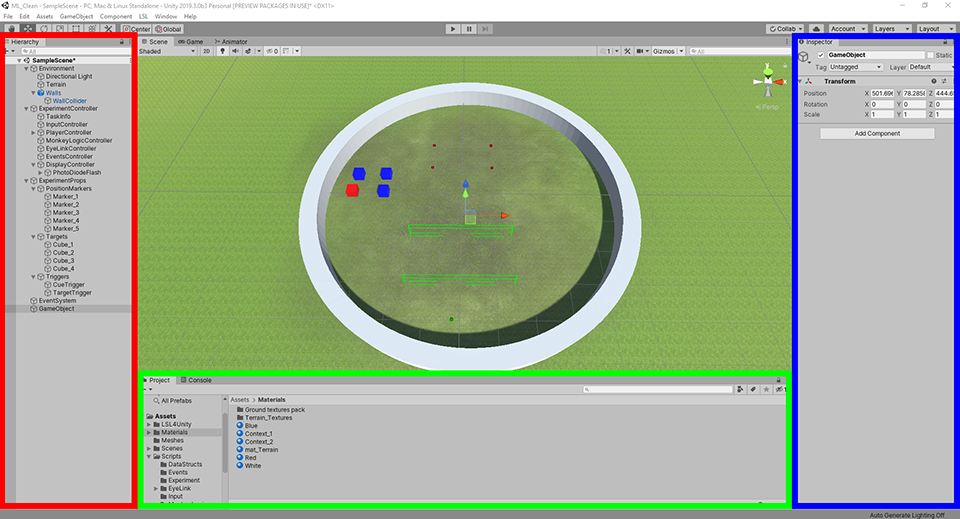
}

If we decided not to call base.Start(); the Human Start function would not be executed when a werewolf is instantiated, therefore it would be nameless and not Billy-Bob.

**Unity Editor**

The above example does not make sense since we would have to define a Start function for each individual human for them to have different properties (e.g. Names). Also, it is unclear how we instantiate these classes. Cue the Unity Editor.

In Unity, each individual “thing” that exists in the scene (e.g. light, player, camera, …) is called a *GameObject*. By default, GameObjects are created with a Name, a Tag and a Transform (i.e. position, rotation and scale). When instantiated, they are assigned a unique InstanceID, which we use to identify individual objects in the game (e.g. collision, gaze target, …). To implement custom behaviors, we attach *Components* to them. Any script that sub-classes the MonoBehaviour class can be attached as a component to a GameObject.



The **Hierarchy** panel lists all the GameObjects in the scene. To attach a component to them you can click the Add Component button of the **Inspector** or drag and drop a script from the **Project** panel.

Any component’s public property can be set via the Inspector, meaning that we could assign our Human script to multiple GameObjects and set each individual name. You can also nest GameObjects under another GameObject, but their Transform components will then be Local to the parent GameObject.

In the SampleTask, GameObjects are nested under 3 groups: Environment, ExperimentController and ExperimentProps. Here they are in more details.

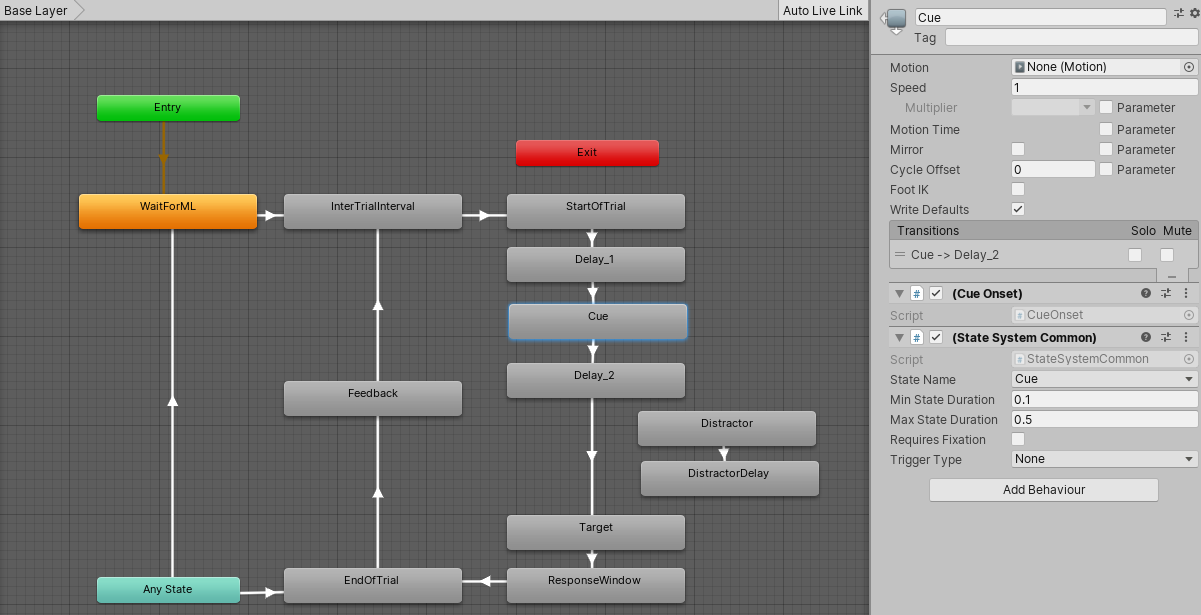
Environment

All the objects not necessary for task progression such as terrain, trees, lights.

Experiment Controller

The Experiment Controller GameObject contains two components: the *ExperimentController* script and the *Animator* controller (i.e. state system).

The *Animator* state system is a graphical state machine in which *conditions* (white arrows) are used to trigger *state* (grey boxes) changes within the Animator. Instead of Components, states receive **Behaviour** scripts, with a default StateSystemCommon script that handles basic state execution (e.g. state name, duration, fixation and trigger requirements). Specific epochs behaviours can also be added but these are mainly used to trigger specific functions within the ExperimentController class, such as: ShowCues(), ShowTargets(), …



The *ExperimentController* (Script) is an abstract class (i.e. needs to be sub-classed) that handles all the inner-workings of the state system and trial progression. It holds references for frame and trial data. It is the core of experiments and gets the specific task information from the Task Info class.

To implement custom tasks, all functions regarding trial progression can be overridden in the experiment controller sub-class. See the These functions are:

protected virtual void Initialize()

This function needs to be called (i.e.: base.Initialize()) from the sub-class. It creates the coroutine responsible for sending frame and trial data to LSL.

public virtual void PrepareAllTrials()

Generates a list of all trials then shuffles it. Calls protected non-overridable functions: GenerateCombinations (i.e. order not important) and GeneratePermutations (i.e. order important). There is more information in the ExperimentController.cs file comments.

public virtual void PrepareTrial()

Increments trial number and places objects (e.g. cue, targets, …) by calling the Prepare[Object] functions. The Prepare[Object], Show[Object] and Hide[Object] are self-explanatory and will not be defined here.

public virtual void PrepareFixationObject()

public virtual void ShowFixationObject()

public virtual void HideFixationObject()

public virtual bool isFixating()

Checks if gaze is on defined fixation object.

public virtual void PrepareCues()

public virtual void ShowCues()

public virtual void HideCues()

public virtual void PrepareTargets()

public virtual void ShowTargets()

public virtual void HideTargets()

public virtual void PrepareDistractors()

public virtual void ShowDistractors()

public virtual void HideDistractors()

public virtual void FreezePlayer(bool ON)

Blocks player movement. The only function with an input argument. ON is a bool variable holding whether the player is frozen (true) or not (false).

public virtual void EndTrial()

Multiple other GameObjects are nested under the ExperimentController. Many of them have no options to modify as the proper behavior is scripted. Here are the ones requiring input.

Task Info

Abstract class containing all de basic task parameters such as trial duration, target objects and cue colors. Can be expanded to add task specific parameters. Keep in mind that the added extra parameters won’t be taken into account in the ExperimentController base class. The specific functions will need to be overridden to access these parameters.

InputController

Simply to select input modality: keyboard, joystick or mouse and their sensitivity.

PlayerController

While there is nothing to modify in the PlayerController, there is a nested FixationPoint GameObject under the PlayerCamera. It is a while sphere to be used as a normal fixation point but the color and shape can be modified here. It is set here to follow the player movement (i.e. children of camera) so it remains constant on screen, but it can be changed to any GameObject in the TaskInfo.

FullScreen View

Under the DisplayController game object, it holds the screen parameters like resolution, offset (whether the right of left display), the player camera FOV, and whether to launch a full screen window on Play. It also contains a nested game object called PhotoDiodeFlash that flashes a square in the bottom left corner of the screen to synch frames with a photodiode. (This section requires testing as it currently changes the square within the full grayscale spectrum on a randomized frame duration. Unsure how sensitive the photodiode will be to these values.)

ExperimentProps

Group containing all the GameObjects referenced in the TaskInfo, such as positions (i.e. the equivalent of PathNodes in UDK), cue and target objects.

PositionMarkers

By default, a GameObject transform is not visible in the Scene view of the Editor, so a DisplayMarker component needs to be added to the object to be visible. Multiple marker types have different colors and this script does not affect task behavior, it’s purely a convenience class to make the objects visible and clickable in the Scene view. For example, setting a marker to “Start” will show a green sphere at the marker position but the object could be used as a target position. The color is just to simplify display. These markers are called *Gizmos* and can be made visible in the Game view. The only thing used for task progression in these objects are the transforms.

Cues, Targets and Triggers

Lists of GameObjects used in the TaskInfo script. Triggers can be used to “trigger” specific state transitions such as Cue and Target onsets in the SampleTask. Other than a DisplayMarker script for triggers no specific behaviors need to be added. Simply make sure they have colliders.

**Other notes:**

Monkeylogic eye calibration

The eye data needs to be calibrated before running the Unity task. The calibration data is then sent to Unity and this allows it to get a raw data sample directly from the Eyelink and properly convert the data to gaze position in both pixels and degrees. This is an overview of the calibration algorithm.

From the timing script we can access the calibration data from: *eye\_.CalFun.tform*, which is a (1,3) cell array of structures containing the useful fields:

* **{1}:** offset
  + [x,y] Offset from the center of the screen
* **{2}:** rotation\_t
  + [2x2] rotation matrix, usually = eye(2)
* **{3}:** tdata.T
  + [3x3] forward projective matrix to convert eyelink raw data to degrees

To convert from eyelink raw data to pixel values we need these formulas:

[X, Y, adjust] = [RawX, RawY, 1] \* eye\_CalFun.tform{3}.tdata.T

EyeDegX = X / adjust

EyeDegY = Y / adjust

([EyeDegX, EyeDegY] – offset) \* rotation\_t

ScreenWidth = eye\_.Screen.Xsize

ScreenHeight = eye\_.Screen.YSize

PixPerDeg = eye\_.Screen.PixelsPerDegree

Assuming that the top-left is (0,0) in pixel values:

EyePixX = (EyeDegX \* PixPerDeg) + 0.5 \* ScreenWidth

EyePixY = - (EyeDegY \* PixPerDeg) + 0.5 \* ScreenHeight

These values are also accessible via:

* MLConfig.EyeTransform{1,1} : offset
* MLConfig.EyeTransform{1,2} : rotation
* MLConfig.EyeTransform{1,3} : tform
* MLConfig.PixelsPerDegree(1)
* MLConfig.Screen.Xsize/Ysize

State system

The order of script execution is Update -> State Machine Update -> Late Update so we send the Player/Eye data on Update to the ExperimentController, then the state system checks the values on state machine update and triggers the state change accordingly.

On the same frame that the condition is met in Update, both the on state exit and on state enter of the next state are executed. The next frame will execute the Update of the second state.

**IMPORTANT**

You need to uncheck the “Has exit time” transition property in the animator states, if not it will add a delay between script execution and state changes. Without it, it will use the value defined in “Exit Time” as a timer and will transition after X number of seconds have elapsed.

Both Transition Duration and Transition Offset need to be set to 0.