**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**

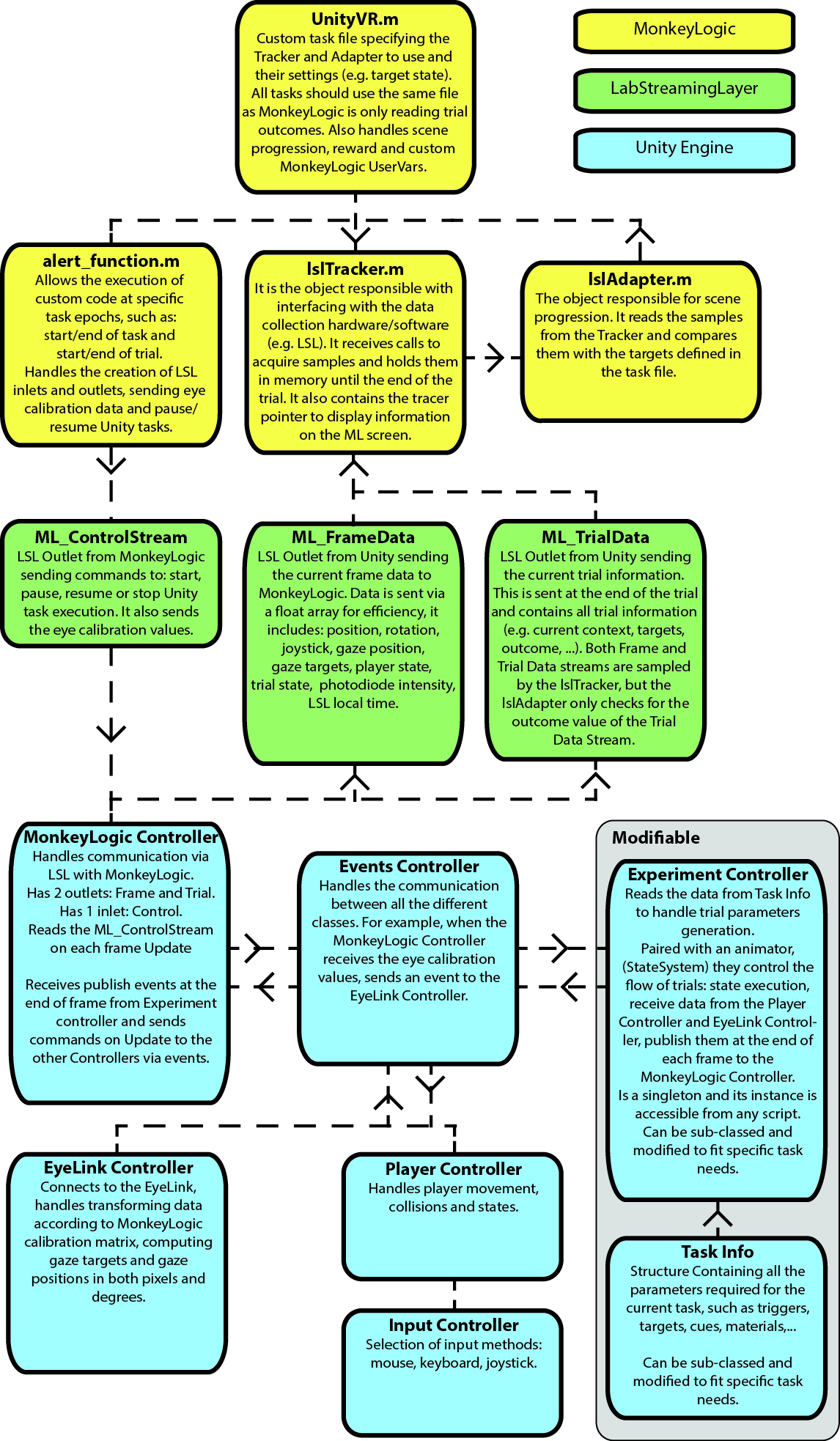
This installs everything needed for the Unity example tasks to work, including the LabStreamingLayer for Unity scripts.

* 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 and positions
* Cue objects and materials
* 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;

}

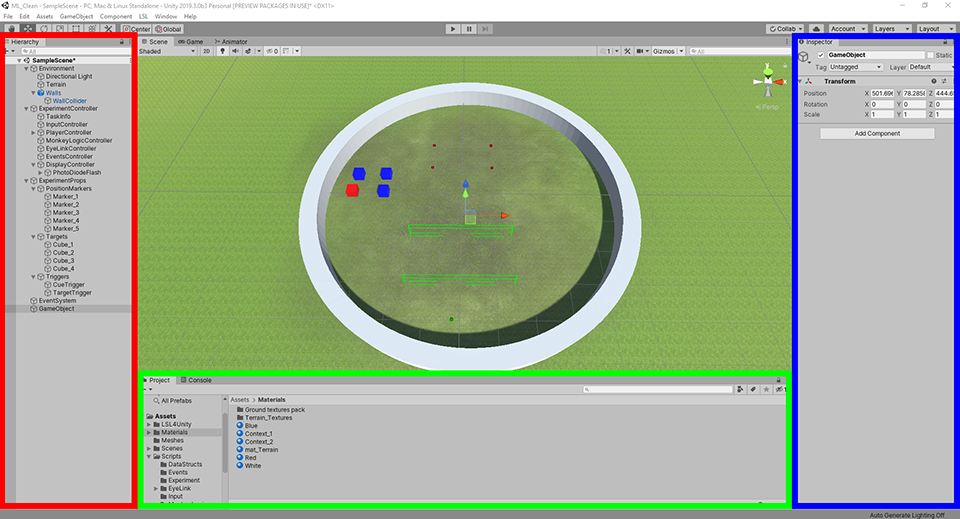
}

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). 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.



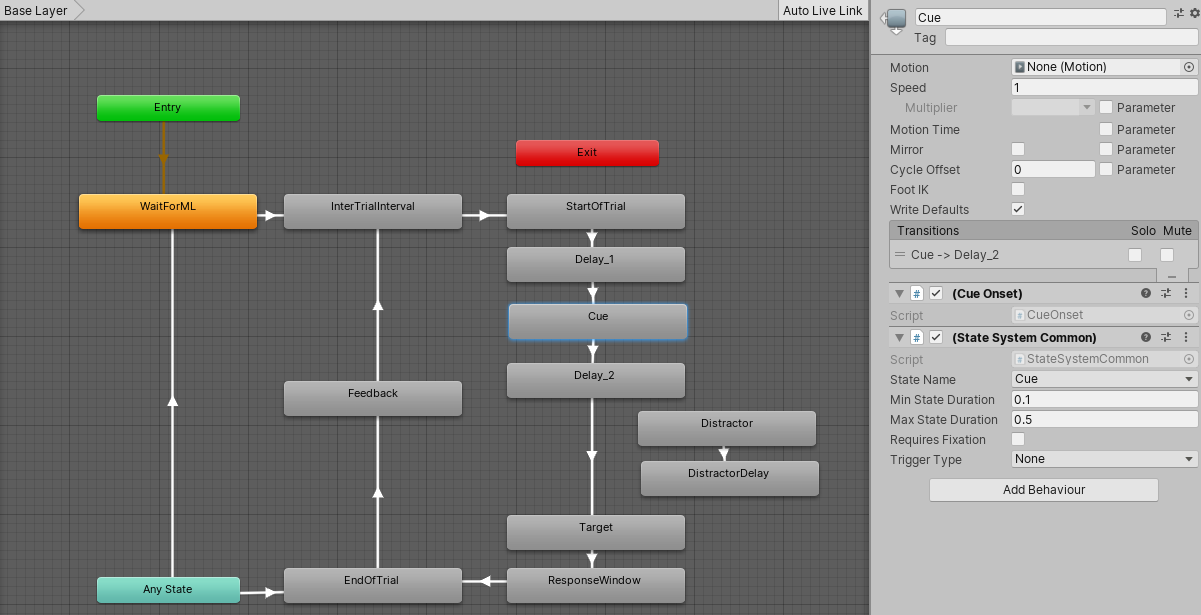
The **Hierarchy** panel lists all the GameObjects in the scene. To attach a component to them you can click the Add Component button of the **Inspector** or drag and drop a script from the **Project** panel.

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.

Experiment Controller (Animator)

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

* The 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(), …



Experiment Controller (Script)

Abstract class

Experiment Controller virtual/abstract functions are:

protected virtual void Initialize()

public virtual void PrepareAllTrials()

public virtual void PrepareTrial()

public virtual void PrepareFixationObject()

public virtual void ShowFixationObject()

public virtual void HideFixationObject()

public virtual bool isFixating()

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)

public virtual void EndTrial()

Task Info

FullScreen View