Multi-Task Unified Suite for Experiments

Developer’s Manual

TABLE OF CONTENTS

1. [Introduction 8](#_Toc149124694)
2. [Purpose 8](#_Toc149124695)
3. [Audience 8](#_Toc149124696)
4. [M-USE Overview 8](#_Toc149124697)
5. [System Setup 9](#_Toc149124698)
6. [Hardware Requirements 9](#_Toc149124699)
7. [Unity 10](#_Toc149124700)
8. [Installing M-USE 10](#_Toc149124701)

[Preparing a Machine for Development 10](#_Toc149124702)

1. [Running M-USE 11](#_Toc149124703)
2. [Configuration Files Setup 11](#_Toc149124704)

[Session Config 12](#_Toc149124705)

[Task Config 14](#_Toc149124706)

[Config File Formats 16](#_Toc149124707)

1. [Resources Folder 17](#_Toc149124708)

[Setting Up the Resources Folder 17](#_Toc149124709)

1. [Creating and Running a Build 17](#_Toc149124710)
2. [Task Buttons Grid – Session Config Implementation 18](#_Toc149124711)
3. [M-USE State Machine 19](#_Toc149124712)
4. [States 19](#_Toc149124713)

[State Initialization 20](#_Toc149124714)

[State Update 20](#_Toc149124715)

[State Termination 20](#_Toc149124716)

1. [Control Levels 21](#_Toc149124717)

[M-USE State Machine 22](#_Toc149124718)

1. [Session Level 22](#_Toc149124719)
2. [Task Level 23](#_Toc149124720)
3. [Trial Level 23](#_Toc149124721)
4. [Transient Levels 23](#_Toc149124722)

[Calibration Level 23](#_Toc149124723)

[Import Settings Level 23](#_Toc149124724)

1. [Task Generation 24](#_Toc149124725)
2. [Generate Task Scripts 24](#_Toc149124726)
3. [Generate Task Scene 24](#_Toc149124727)
4. [Generate Task Namespace 24](#_Toc149124728)
5. [Define the TaskDef Class 25](#_Toc149124729)
6. [Define the BlockDef Class 25](#_Toc149124730)
7. [Define the TrialDef Class 26](#_Toc149124731)
8. [Define the StimDef Class 26](#_Toc149124732)
9. [Setting Up Configuration Folder & Files 27](#_Toc149124733)
10. [Setting Up Your Resources Folder & Files 27](#_Toc149124734)
11. [Defining the Task Level 27](#_Toc149124735)
12. [Defining the Trial Level 29](#_Toc149124736)
13. [Using the State System in a Task 29](#_Toc149124737)

[AddInitializationMethod 29](#_Toc149124738)

[AddUpdateMethod 30](#_Toc149124739)

[SpecifyTermination 30](#_Toc149124740)

[AddTimer 30](#_Toc149124741)

[AddDefaultTerminationMethod 31](#_Toc149124742)

[AddUniversalTerminationMethod 31](#_Toc149124743)

1. [M-USE Modules 32](#_Toc149124744)
2. [Experimenter Display 32](#_Toc149124745)

[Initializing the Experimenter Display 32](#_Toc149124746)

[Trial Info Panel 33](#_Toc149124747)

[Block Info Panel 33](#_Toc149124748)

[Session Info Panel 33](#_Toc149124749)

[Log Info Panel 34](#_Toc149124750)

[Hot Key Panel 34](#_Toc149124751)

[Config UI Panel 35](#_Toc149124752)

[Player View Panel 35](#_Toc149124753)

1. [Feedback Controllers 36](#_Toc149124754)

[Token Feedback 36](#_Toc149124755)

[Slider Feedback 36](#_Toc149124756)

[Halo Feedback 37](#_Toc149124757)

[Touch Duration Feedback 37](#_Toc149124758)

[Audio Feedback 38](#_Toc149124759)

1. [UI Classes 38](#_Toc149124760)

[HumanStartPanel 38](#_Toc149124761)

[USE\_StartButton 39](#_Toc149124762)

1. [Eye Tracking 39](#_Toc149124763)

[Setting Up an Eye Tracker 39](#_Toc149124764)

[TobiiEyeTrackerController 39](#_Toc149124765)

[TobiiGazeDataSubscription 40](#_Toc149124766)

[Setting Up a Gaze Selection Handler 40](#_Toc149124767)

1. [Sync Box Controller 40](#_Toc149124768)

[Setting Up a Sync Box 41](#_Toc149124769)

1. [Event Code Controller 41](#_Toc149124770)

[Setting Up Event Codes 41](#_Toc149124771)

1. [Serial Port Controller 41](#_Toc149124772)

[Setting Up a Serial Port 41](#_Toc149124773)

1. [Input Broker 41](#_Toc149124774)

[Using the Input Broker 41](#_Toc149124775)

1. [Input Tracker 41](#_Toc149124776)

[Using an Input Tracker 41](#_Toc149124777)

[Mouse Tracker 41](#_Toc149124778)

[Gaze Tracker 41](#_Toc149124779)

[Joystick Tracker 41](#_Toc149124780)

1. [Selection Handler 41](#_Toc149124781)

[Creating A Selection Handler 42](#_Toc149124782)

[Using the Selection Handler 42](#_Toc149124783)

1. [Resource Management 43](#_Toc149124784)
2. [Stimuli Management 43](#_Toc149124785)

[Stim Groups 43](#_Toc149124786)

[Stim Defs 43](#_Toc149124787)

[Using Stimuli in a Task 43](#_Toc149124788)

[Generating Stimuli in Blender 44](#_Toc149124789)

1. [Context Management 44](#_Toc149124790)

[Generating Contexts 44](#_Toc149124791)

[Preparing Distinct Context Conditions 44](#_Toc149124792)

[Interfacing with Dalle2 in Python 46](#_Toc149124793)

[Using Contexts in a Task 47](#_Toc149124794)

1. [Data Management 48](#_Toc149124795)
2. [Session Data 48](#_Toc149124796)
3. [Block Data 48](#_Toc149124797)

[The BlockData class is responsible for managing data related to experiment blocks. It records information such as the subject's ID, age, session timestamp, the name of the active task, block count, and additional statistics like the number of reward pulses and aborted trials within a block. This data is crucial for tracking the progress of the experiment at the block level. 48](#_Toc149124798)

1. [Trial Data 48](#_Toc149124799)
2. [The TrialData is a class dedicated to collecting and storing data associated with individual experiment trials. It captures data such as the subject's ID, age, session timestamp, task name, block count, trial count within a task, and any abort codes associated with trials. This class helps in tracking the specifics of each trial conducted during the experiment. 48](#_Toc149124800)
3. [Frame Data 48](#_Toc149124801)
4. [Gaze Data 48](#_Toc149124802)
5. [The GazeData is responsible for capturing gaze-related data throughout the experiment. It records subject-related information such as ID and age, session timestamp, task name, block count, trial count within a task, the current trial state, frame count, and frame start time in Unity. Additionally, it collects gaze-specific metrics such as pupil diameter and gaze point coordinates. This data is valuable for analyzing subjects' eye movements and visual attention during the experiment. 48](#_Toc149124803)
6. [Serial Data 48](#_Toc149124804)
7. [The SerialSentData and SerialRecvData are designed to manage data related to serial communication. The SerialSentData stores data sent through the serial port, while the SerialRecvData records data received via a serial port. These classes enable the recording and analysis of data exchanged between the M-USE software and external hardware or devices via serial communication. 49](#_Toc149124805)
8. [Summary Data 49](#_Toc149124806)
9. [The TaskSummaryData class is responsible for summarizing and storing experiment data. It manages the creation of folders for data storage, handles data storage options (local or server), and facilitates the addition of task run data to summary files. This class ensures that experiment data is organized, stored, and retrievable for further analysis or reporting purposes. 49](#_Toc149124807)
10. [Server Management 49](#_Toc149124808)

[ServerManager Class 49](#_Toc149124809)

[Server Setup 49](#_Toc149124810)

[Connecting to Server During a Session 50](#_Toc149124811)

1. [Troubleshooting 52](#_Toc149124812)
2. [Debugging Approaches 52](#_Toc149124813)

[Using the Log File 52](#_Toc149124814)

[Debug Statements 52](#_Toc149124815)

[Using the Unity Profiler 53](#_Toc149124816)

# Introduction

## Purpose

The Multitask Universal Suite for Experiments (*M-USE*) is a plugin for the Unity video game engine that assists with the development and control of behavioral neuroscience experiments. As the name implies, it has a particular focus on *multiple* tasks, in two ways: it supports running tasks in many different experimental setups, and it supports running multiple tasks in a single experimental session. This makes it a powerful tool for between-group and within-subject comparisons.

## Audience

This document is intended as a reference for M-USE *developers*: individuals who are either editing the core functionality of M-USE, or developing new experimental tasks within it. It provides a reasonably high-level overview of the entire suite, explaining how it’s different components operate and interact, including most commonly used classes and methods, but is not intended to provide a complete account of every method in the suite (we are in the process of developing such a reference as well). There is a corresponding *M-USE Experimenter Manual* that is intended as a reference for individuals running experiments with the suite.

# M-USE Overview

M-USE builds consist of four main components: Unity’s native processes, a *hierarchical finite state machine*, a *task library*, and variety of *modules* (see Figure 1). The state machine (see [M-USE State Machine](#_M-USE_State_Machine)) is the “engine” of the build, with almost all operations stemming from methods called from one of its *control levels* (see [Levels](#_Control_Levels)). The top level of the state machine is the *session level* (see [SessionLevel](#_Session_Level)), and anything an M-USE build does can ultimately be traced back to a command called at the session level. Some of the most important of these commands active subordinate control levels, in particular Task and Trial control levels (see [TaskLevel](#_Task_Level) and [TrialLevel](#_Trial_Level)) that govern the operation of particular experimental tasks, and are stored in the task library. When a particular experimental task is started, its Task and Trial levels are added to the state machine, and when it is finished, they are removed from the state machine, and remain inert in the Task Library, ready to load again if needed.

The state machine is entirely encapsulated by the modules, such that all interactions with Unity’s native processes, or other CPU processes (including ones that connect to outside hardware) are mediated through these modules. For example, in order to detect a button press, the InputBroker module’s GetButtonDown method mirrors the output of Unity’s native GetButtonDown method, which itself interacts with other CPU processes to obtain the button’s status.



Figure 1. The main components of an M-USE build, including their basic interactions with the rest of the experimental computer and the external world. Note the encapsulation of the state machine and task library from all other processes, via the intermediary of the M-USE modules.

The remainder of this document is, in essence, an expansion of the previous two paragraphs, with enough detail to allow developers to understand the different elements of the suite.

## System Setup

## Hardware Requirements

To ensure optimal performance and a seamless experience while running MUSE, its essential to meet or exceed the following hardware specifications:

1. Processor (CPU): A multi-core processor with a clock speed of at least 2.5 GHz or higher.
2. Graphics Card (GPU): A dedicated graphics card with DirectX11 or OpenGL 4.5 support is required. For optimal performance with the latest graphics features, a GPU from NVIDIA or AMD with at least 2GB of VRAM is recommended.
3. Memory (RAM): A minimum of 8GB of RAM is recommended for running MUSE.
4. Storage: We recommend you have at least 20GB of free disc space available for installing MUSE.
   1. MUSE Repository: 5.5 GB.
   2. Configs and Resources folders: 252 MB.
   3. MUSE Build: 302 MB.
   4. Session Data: Can be several Gigabytes for full sessions with multiple tasks.
5. Operating System: MUSE is compatible with Windows and MacOS.

## Unity

Install the Unity Hub and the Unity Editor (version 2021.3.31f1) at <https://unity3d.com/get-unity/download>.

## Installing M-USE

### Preparing a Machine for Development

To prepare a machine to run M-USE, install the following applications:

1. Install Git at <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.
2. Install the Unity Hub and the Unity Editor (version 2021.3.31f1) at <https://unity3d.com/get-unity/download>.
3. Install Visual Studio at <https://visualstudio.microsoft.com/downloads/>. To use another text editor or integrated development environment, download it and set it as the default script editor in Unity (tutorial at <https://learn.unity.com/tutorial/set-your-default-script-editor-ide#62a054a2edbc2a0ff716bf04>).

To install **M-USE**:

1. Download the repository from <https://github.com/Multitask-Unified-Suite-for-Expts/M-USE> onto your machine.
2. Open the Unity Hub, click the dropdown arrow on the top right and select “Add project from disk”, then navigate to and select the repository folder you downloaded in step 1.
3. Install Newtonsoft\_Json:
   1. In the Unity editor, navigate to Window > Package Manager and select Packages: Unity Registry from the dropdown and search for Newtonsoft.Json and install the package. Ensure you are searching in their registry and not the packages already in your project.

# Running M-USE

First, download the M-USE repository, as detailed in [Installing M-USE](#_Installing_M-USE).

Before creating and running a build, you first need to obtain and update the Config and Resources folders.

## Configuration Files Setup

*Note: To generate Session Config folders using MATLAB, refer to the ExperimenterManual document.*

A screenshot of a computer

Description automatically generatedDownload the MUSE folder at [http://m-use.psy.vanderbilt.edu/downloads](http://m-use.psy.vanderbilt.edu/downloads/), and place it on your desktop. The MUSE folder contains a Configs folder, a Resources folder, and a Build folder.

The Configs folder contains a SessionConfig file, a SessionEventCodeConfig file, and the configuration folders for each Task.

A screenshot of a computer

Description automatically generated

### Session Config

Inside the MUSE/Configs folder, update the Session Config file for your session. Below is an example of a formatted file. Ensure you update the “ContextExternalFilePath” and “TaskIconsFolderPath” paths to match the location on your machine.

**SessionConfig\_singleType.txt**

OrderedDictionary<string,string> TaskMappings {EffortControl:EffortControl }

Dictionary<string,string> TaskIcons {EffortControl:EffortControl}

string ContextExternalFilePath "~//MUSE//Resources//Contexts"

string TaskIconsFolderPath "~//MUSE//Resources//TaskIcons"

bool IsHuman false

bool MacMainDisplayBuild true

bool SyncBoxActive false

bool EventCodesActive true

bool GuidedTaskSelection false

//Optional - Task Button Grid Specification

List<int> TaskButtonGridSpots {0, 3, 6, 9, 12, 15}

// SyncBox Specifications

string SerialPortAddress "\\\\.\\COM3"

int SerialPortSpeed 115200

List<string> SyncBoxInitCommands {"INI", "ECH 0", "TIM 0", "LIN 33", "LVB 0", "NSU 2", "NPD 10", "NHD 2", "NDW 16", "CAO 20000", "TBP 1000", "TBW 50", "TIB 1", "LOG 1"}

int SplitBytes 2

int RewardHotKeyNumPulses 1

int RewardHotKeyPulseSize 250

// Shotgun Handler Specifications

float ShotgunRaycastCircleSize\_DVA 1

float ParticipantDistance\_CM 60

float ShotgunRaycastSpacing\_DVA 0.3

|  |  |  |
| --- | --- | --- |
| *Variable Name* | *Type* | *Description* |
| TaskMappings | OrderedDictionary<string,string> | Depicts the relationship between the name of the Config folder and the tasks within the suite. The first half of each entry is the task name, and the second half is the Config folder name for that task. (Ex: EffortControl:EffortControl). To run multiple tasks, separate each entry with a comma. |
| TaskIcons | OrderedDictionary<string,string> | Depicts the relationship between the name of the Config folder and the task icon image to be used in the TaskSelection Scene. The first half of each entry is the task name, and the second half is the name of the task icon image file located in MUSE/Resources/TaskIcons. (Ex: EffortControl: EffortControl). To run multiple tasks, separate each entry with a comma. |
| ContextExternalFilePath | string | The path to the folder containing the Context PNGs that are used in the session.  Note: You will need to update this path to match the location on your machine. |
| TaskIconsFolderPath | string | The path to the folder containing the Task Icon PNGs that are used in the session.  Note: You will need to update this path to match the location on your machine. |
| IsHuman | bool | Dictates whether a human participant is running the session. Set to true if you wish to use the human version. |
| MacMainDisplayBuild | bool | Dictates whether the session is being run on a Mac. Set to true if a Mac is being used. |
| SyncBoxActive | bool | Dictates whether a Sync Box is being used for the session. Edit the SerialPortAddress, SerialPortSpeed, SyncBoxInitCommands, SplitBytes, RewardHotKeyNumPulses, and RewardHotKeyPulseSize variables for your Sync Box, as necessary. Set to true if you’ve connected and are using a Sync Box device to issue reward and timing pulses. |
| EventCodesActive | bool | Dictates whether Event Codes are being used for the session. Set to true if Event Codes are being used. |
| GuidedTaskSelection | bool | Dictates whether task selection is restricted to the order listed in the TaskMappings. Set to true to enable controlled task selection in the specified order. |
| TaskButtonGridSpots | List<int> | Manually specified grid locations for the task buttons. There are 20 total grid spots (0-19).  See [TaskButtonsGeneration](#_Task_Buttons_Grid) for more info. |
| ShotgunRaycastCircleSize | float | Default value of 1.25 is assigned if not specified. Assigns the radius of the shotgun selection that is used for the ShotgunSelectionHandler. Default value of 1.25 is assigned if not specified. |
| ParticipantDistance\_CM | float | Assigns the participant’s distance from the display area for the ShotgunSelectionHandler. Default value of 60 is assigned if not specified. |
| ShotgunRaycastSpacing\_DVA | float | Assigns the spacing of the Shotgun Raycast for the ShotgunSelectionHandler. Default value of 0.3 is assigned if not specified. |
| MaxStimLoadingDuration | Float | Used to control the max amount of time for a stim to load, whether locally or from the server.  Default value is 2. Minimum value allowed is 1. |

### Task Config

Next, we will create/update the individual task’s Configuration Folder & files. Each task needs an individual Task Config folder inside the MUSE/Configs folder, where its configuration files are stored. If you’re creating a new task, create its task config folder by copying an existing task’s config folder and updating the folder and file names.

A screenshot of a computer

Description automatically generated

TaskDef Config

The TaskDef specifies values for the task and must be a tab-delimited file (tabs separate each column). An example TaskDef for the Effort Control task is shown below.

EffortControl\_TaskDef\_singleType.txt

List<string>    FeedbackControllers    ["Audio", "Token"]

Vector3    ButtonPosition (0, 0, 0)

float  ButtonScale    1.2

float  TouchFeedbackDuration  0.3

BlockDef Config

The BlockDef specifies values for each block within the task and must be a tab-delimited file (tabs separate each column). The first row contains the classes fields, and each subsequent row contains an instance of the class. An example BlockDef for the Effort Control task is shown below.

EffortControl\_BlockDef\_array.txt

BlockName   NumTrials  NumClicksLeft  NumClicksRight NumCoinsLeft   NumCoinsRight  ClicksPerOutline   NumPulsesLeft  NumPulsesRight PulseSizeLeft  PulseSizeRight ContextName

EC3\_2-EC1\_8    1  3  2  3  1  1  2  8  500    500    011\_058\_004\_001\_3

EC3\_8-EC1\_8    1  4  2  3  1  1  8  8  500    500    011\_058\_004\_001\_3

EC3\_2-EC3\_8    1  3  4  3  3  1  2  8  500    500    011\_058\_004\_001\_3

TrialDef Config

The TrialDef specifies values for each trial within the task and must be a tab-delimited file (tabs separate each column). The first row contains the classes fields, and each subsequent row contains an instance of the class, formatted similarly to the BlockDef above. Effort Control does not include TrialDef Config, and instead generates TrialDefs given the BlockDef and NumTrials.

StimDef Config

The StimDef specifies the Quaddles that are used within the task. The first row contains the classes fields. Effort Control does not include StimDef Config, but below is the format from the Visual Search task.

VisualSearch\_StimDef\_array.txt

StimIndex   FileName   PrefabPath

0  Stim1.gltf   ""

1  Stim2.gltf ""

2  Stim3.gltf  ""

**2D Stim vs 3D Stim Changes:**

Note: Using 2D Stim (.PNG files) and using 3D Stim (.GLTF files) require different stim sizes and positions. If you’re switching between 2D and 3D stim, you will need to go into any applicable Task’s TaskDef configuration files to adjust the “ExternalStimScale” variable, as well as adjusting any positions specified inside the Task’s BlockDef and TrialDef configuration files. For example, the Continuous Recognition task’s ExternalStimScale variable is set to 0.5 for .GLTF files and 3.0 for .PNG files.

ConfigUiDetails

The ConfigUiDetails specifies specify variables that will be presented (and configurable) on the Experimenter Display’s User Interface.

EffortControl\_ConfigUiDetails\_json.json

{

 "varsNumber": {

   "inflateDuration": {

     "name": "Inflate Duration",

     "value": 45,

     "min": 30,

     "max": 90,

     "precision": 1,

     "isRange": 1,

     "hidden": 0

   },

   "itiDuration": {

     "name": "ITI Duration",

     "value": 0.5,

     "min": 0.1,

     "max": 1,

     "precision": 1,

     "isRange": 1,

     "hidden": 0

   }

},

 "varsBoolean": {},

 "varsString": {}

}

EventCode Config

The EventCode config specifies events that are sent across the Sync Box.

EffortControl\_EventCodeConfig\_json.json

{

 "BalloonChosen": {

   "Value": 12026,

   "Description": "Balloon Chosen"

 }

}

### Config File Formats

There are 3 types of Configuration files: SingleType, Array, and JSON. Each configuration file needs to include its type in the file name, with a preceding underscore (ex. SessionConfig\_singleType.txt).

SingleType

This type is for configuration files that produce a single instance of a particular type (see **TaskDef**). Each row is a variable, and must be presented in the following format, separated by tabs:

“VariableType *TAB* VariableName *TAB* VariableValue”.

Array

This type is for configuration files where the first row contains the fields (headers) of the class, and each subsequent row is an instance of the class (see **BlockDef**). After being read in, the output is an array of a particular type and must be presented in the following format, separated by tabs:

“VariableType *TAB* VariableName *TAB* VariableValue”.

JSON

For configuration files in JSON format that produce a single instance of a single type (see **ConfigUiDetails**).

## Resources Folder

### Setting Up the Resources Folder

If you haven’t already downloaded the MUSE folder (as detailed in [ConfigurationFileSetup](#_Configuration_File_Setup) above), Download the **MUSE** folder from [http://m-use.psy.vanderbilt.edu/downloads](http://m-use.psy.vanderbilt.edu/downloads/) and place it on your desktop.

The Resources folder is located within the **MUSE** folder, and contains folders for Contexts, Task Icons, and Stimuli.

Contexts

The Contexts folder contains all .PNG image files to be used as backgrounds for the tasks. Here you can add any .PNG files you wish to use for your tasks.

Task Icons

The Task Icons folder contains all .PNG image files to be used as the icons for the tasks in the TaskSelection scene. Add a task icon .PNG image file for each new task you create. Use the task’s name as the name of the .PNG file, and ensure it matches the task’s dictionary entry in the SessionConfig’s TaskIcons variable.

Stimuli

The Stimuli folder contains all .GLTF Quaddle files to be used as the stimuli for the tasks. Any tasks using stimuli will specify which stimuli will be loaded in their respective StimDef file, and the stim they specify must correspond to a .GLTF file listed here in the Stimuli folder.

Now, with the repository downloaded, the Configs and Resources folders updated, you’re ready to create and run a build in Unity.

## Creating and Running a Build

To create and run a Unity Build:

1. In the Unity Editor, navigate to File > Build Settings.
2. Add the scene’s you wish to include in the build to the Scenes in Build section at the top of the build settings.
3. Select your Build Platform (Windows/Mac/Linux, WebGL, etc.).
   1. Note: If you’re going to create a WebGL build, you need to delete the StandaloneFileBrowser folder located at Assets/\_Scripts/M\_USE/M\_USE Modules/InitializationScreen/LocateFile/StandaloneFileBrowser.
      1. Conversely, if you’re going from a WebGL build to a Windows/Mac/Linux build, you will need to add the StandaloneFileBrowser folder back into the project.
4. Click Build and select a folder location for the build to be created in.
5. Unity will compile the project and its assets into a standalone executable inside the folder location you specified in step 3.
6. Double click the executable file (.exe) to run the build.

## Task Buttons Grid – Session Config Implementation

This section details the “TaskButtonGridSpots” session config variable and how to properly implement it into the Session Configuration file.

Background:

A Task Button is displayed for each task during the Task Selection scene. We’re using Unity’s UI Grid Layout System to create a centered grid for which you’re able to specify exact grid locations for each task button.

Implementation into the Session Config:

To specify exact grid spots for each task, put the following TaskButtonGridSpots variable into your session config file, with specified grid spots for each task.

List<int> TaskButtonGridSpots {0, 3, 6, 9, 12, 15}

There are 20 total grid spots automatically generated, and thus the grid spots start at 0 and go to 19. The order of the values inside the TaskButtonGridSpots variable is used in conjunction with the order of the tasks in the TaskMappings variable in the Session Config file. For example, if my TaskMappings variable is {CR:CR, EC:EC, WM:WM} and I want CR to be at grid spot 2, EC to be at grid spot 7 and WM to be at grid spot 14, the TaskButtonGridSpots value needs to be {2, 7, 14}.

Note: Specifying grid locations for the task buttons is optional, and if the TaskButtonGridSpots variable is omitted from the Session Config file, the buttons will populate on the grid (starting at spot 0), in the order they’re specified in the TaskMappings variable.

# M-USE State Machine

The hierarchical finite state machine that governs the operation of M-USE is composed of a number of *control levels*, each of which is always in one, and only one, *state*. Here we define the general notions of states and control levels, before turning to the specific states and levels that make up the M-USE finite state machine.

## States

States consist of *initialization methods* that occur once, on the first frame they are active; *update*, *fixedupdate*, and *lateupdate* methods that occur as part of Unity’s internal update loop (see <https://docs.unity3d.com/Manual/ExecutionOrder.html> for an overview of this loop); *termination criteria* that specify the conditions that will stop the states; *termination methods* that occur once, immediately after a termination criterion is true; and a designation of the successor state that will become active after the current state’s termination, or a null designation if no state should follow, which will terminate the control level (Figure 1). All of these, with the exception of at least one termination criterion and associated successor state, are optional, thus the minimal possible state consists of a single termination criterion, in which case the state would do nothing at all until this criterion was passed, at which point it would terminate and pass control to its successor state.

Figure 2. The complete internal logic of a single state. Dashed lines indicate optional methods - thus the only strictly required element of a state is a single Termination Criterion with an associated successor state.

### State Initialization

On the first frame in which a state is active, initialization methods, if they exist, are run. There are three types of initialization methods: universal, specific, and default, all of which are optional (thus it is possible to have a state with no initialization methods at all). Universal methods are run first. These will be followed by any specific initialization methods that are currently active. If there are no specific initialization methods active, default initialization methods will be run, if they exist.

If a state has only one initialization method (in our experience, the most common situation), there is no need for universal or default methods, which exist simply to avoid copying and pasting code into multiple specific initialization methods. A universal initialization method is equivalent to copying and pasting the same code at the start of every specific initialization method, while a default initialization method is the same as replacing a number of specific initialization methods with identical code.

These methods are specified using State.AddUniversalInitializationMethod(), State.AddSpecificInitializationMethod(), and State.AddDefaultInitializationMethod(). If no default method is specialized, but a specific initialization method is provided, the specific method will be treated as default, and run on every initialization of the state (this should probably eventually be refactored as well).

After initialization, the state moves immediately onto its update cycle.

### State Update

State updates are the main way in which the M-USE state system is directly tied to Unity’s own internal frame-locked logic. Every frame in which a state is active, any Update Methods, if they exist, will run exactly once, called by the UnityEngine.Update() method. Following this, any FixedUpdate methods, if they exist, will be called by the UnityEngine.FixedUpdate() method, as many times as Unity deems necessary to maintain its physics timing (determined by Unity’s timescale parameter). Following the FixedUpdate loop, any LateUpdate methods, if they exist, will be called by the UnityEngine.LateUpdate() function.

These different state update methods are specified by the State.AddUpdate(), State.AddFixedUpdate(), and State.AddLateUpdate() methods, respectively.

After a pass through the three update methods as just described, the termination criteria are evaluated as the final act of the state during UnityEngine.FixedUpdate(). If any of these criteria are met, the state will run the appropriate termination methods and terminate itself.

### State Termination

All states must have at least one termination criterion specified, which is a boolean evaluated after each Update cycle. After the first termination criterion is triggered, the update cycle ends, subsequent termination criteria are not evaluated, and appropriate termination methods can run. As with the initialization methods, there are *universal*, *specific*, *default*, and *universal late* state termination methods. Universal methods, if they exist, are run first. If the triggered termination criterion has an associated specific termination method, it will be run after the universal termination methods, if not, any default termination methods will be run after the universal termination methods. If there are any universal late termination methods, they will be run after the specific or default methods.

As with the initialization methods, universal and default methods are essentially a means of preventing unnecessary copy/pasting of code, however in our experience it is much more common to need multiple termination methods, and so the distinction between universal and default becomes more relevant to developers. Using a universal method is the same as adding the same methods to the start of every specific termination method (or at the end of the specific method for universal late methods). Using a default method is the same as making a subset of these specific termination methods identical. In the case where a state has only one termination specified, there is no need to make a distinction between universal, default, and specific terminations.

State.AddUniversalTerminationMethod(), State.AddDefaultTerminationMethod() and State.AddUniversalLateTerminationMethod() add the corresponding method types to a state. Since specific termination methods must be tied to a specific termination criterion and a successor state, there is no corresponding State.AddSpecificTerminationMethod(). Rather, various overloads of State.SpecifyTermination() are used, all of which include arguments specifying a termination criterion and successor state, have optional arguments adding specific termination methods to be triggered by the specific criterion, as well as optional arguments setting the active initialization method of the successor state. The successor state can itself be determined at runtime, in which case its argument is of class Func<State> instead of just State.

Because there are many cases in which the termination criterion consists in checking if a certain amount of time has passed since the beginning of the state, State.AddTimer() can be used with a float as its first argument, which indicates the desired maximum duration of the state in seconds.

After all termination methods are run, the state is finished, and control passes on the subsequent frame to the successor state, or if the successor is specified as null, then the control level housing this state will itself begin its termination process.

## Control Levels

A control level consists of a number of state definitions, and the rules governing the transitions between them, as well as its own initialization and termination conditions, that will run on the first and last frame that the level is operational (Figure 1B). Control levels also have their own initialization and termination methods, that are run on the first and last frames, respectively, in which the level is operational.

The *hierarchical* aspect of the state machine comes from the fact that a given state can have a *child* control level, which will operate so long as the state is active. For example, the Session Level’s *RunTask* state has a child *Task* level, which itself has RunBlock state with a child *Trial* level (Figure 2).

Control levels do not have to always run through the same sequence of states in the same order. A given state can have many termination criteria, each of which leads to a different successor state. This allows for a great deal of flexibility in their activity, including the activation of transient states that only become active in particular circumstances. For example, if an experimenter notices that eyetracking performance of a participant is poor, they can press a hotkey that ends the active Task’s current state and starts its EyetrackerCalibration state, which has its own child level that runs a calibration routine. Once this routine is done, the child level terminates, and the EyetrackerCalibration state ends, passing control back to the regular Task level states.

### M-USE State Machine

M-USE’s state machine (Figure 2) always has an active Session Level, which ultimately governs virtually all M-USE activity by activating child levels. The most critical of these are the Task and Trial levels, which together define the operation of a task. A particular task is chosen during the Session Level’s SelectTask state, activated during its LoadTask state, and the Task level then becomes a child of the RunTask state. The Trial level is the child of the Task level’s RunBlock state. The primary work in developing a new task usually consists of defining the custom Trial level states .



Figure 3. An illustration of the M-USE state machine highlighting the most critical inheritance relationships between states and control levels. Larger white rectangles are control levels, smaller grey rectangles are states. States with solid outlines cannot be modified, while those with dotted outlines can have additional methods added by task developers. The primary “action” of M-USE takes place during the Session Level’s RunTask state, during which a particular custom task is given control.

Each of the Session, Task, and Trial levels incorporates a conditional looping structure defined by different state termination criteria, described in its corresponding section below.

## Session Level

The Session level begins with an InitScreen state, in which experimenters specify participant and session IDs, select the source of configuration files for this session, and select the destination for data files to be written. This is followed by a SetupSession state, which during its own SetupSession\_Level child level, imports the SessionConfig and EventCodeConfig files from the location specified during InitScreen, creates the main data folder, and then loops through every task specified in the SessionConfig’s TaskMappings field, verifying that its task-specific configuration files can be read and processed to generate appropriate trial definitions. This allows many possible task-configuration-specific errors to be isolated prior to starting the rest of the session.

SelectTask follows SetupSession. Here, if there are more tasks remaining to run this session (defined by the TaskMappings variable in the SessionConfig file) the participant selects one of these tasks. If no tasks remain to be run, SelectTask terminates immediately and the next state is FinishSession. The corresponding task scene is loaded during LoadTask, its configuration files are re-imported and processed during SetupTask, and its Task Level is made the child level of the RunTask state. During RunTask itself, control over most aspects of the suite is passed to the Task Level and its corresponding Trial Level.

After RunTask, the TaskLevel loops back to SelectTask, to see if any more tasks remain to be selected. If not, FinishTrial is made active, and various housekeeping tasks are performed (e.g. finishing writing all data files, closing serial connections, etc).

## Task Level

Each run through the main task level loop constitutes a single block of trials. By the time the Task Level starts, it has already been populated with a list of at least one BlockDef, which itself has a list of at least one TrialDef. These are generated from the information in various configuration files during the HandleTrialAndBlock state of the VerifyTask\_Level. The Task Level’s RunBlock state, during its initialization methods, selects the current BlockDef, and assigns its list of TrialDefs to the Trial Level. Once the trial level is finished running through the list of trials in a block, it terminates and the BlockFeedback state shows desired feedback. If there are still blocks remaining to be run, the level loops back to the RunBlock state, otherwise it moves onto the FinishTask state, performs end-of-task housekeeping, and finishes the Task Level.

## Trial Level

Each run through the main trial level loop constitutes a single trial. The LoadTrialStims and SetupTrial states should probably be combined into one state, but they iterate trial counters, prepare timing variables, and load the StimGroups that will be used during the trial. SetupTrial is followed by one of the task’s custom trial state, which continue until FinishTrial, a state whose primary purpose is to evaluate the current status of the block (has performance reached a termination threshold, are there more potential trials to run, etc). If there are trials remaining, FinishTrial loops back to LoadTrialStims, otherwise the block is finished and the TrialLevel terminates, passing control back to the Task Level.

## Transient Levels

### Calibration Level

Details need to be added

### Import Settings Level

The ImportSettings level is a child of both the SetupSessionLevel.ImportSessionSettings state, and the VerifyTaskLevel.ImportTaskSettings state. In both cases, it is passed a list of class SettingsDetails, which includes fields specifying the folder to search, the search string, and the type to cast the parsed file to. Once a file has been found and read as a pure text string, it is passed to the desired type.

# Task Generation

The following task generation steps use the Effort Control task as an example.

## Generate Task Scripts

In the Unity editor, click the USE button in the top, horizontal navbar and select Create Task Scripts. This will create a new task folder inside of \_USE\_Tasks and will generate the task’s default TrialLevel, TaskLevel, and Namespace scripts.

## Generate Task Scene

In the Unity editor, click the USE button in the top, horizontal navbar and select Create Task Scene. Input the same name that you used above when creating the scripts.

**Camera Setup:**

In your scene, name your main camera GameObject “YourTaskName\_Camera”, replacing YourTaskName with the name of your task. This is crucial, as it allows our scripts to find your scene’s Camera GameObject.

**Canvas Setup:**

If you plan to use UI elements in your task, you will need to setup a Canvas in your scene. Follow these steps to ensure its compatibility with the Suite:

1. In your scene, right click and select UI/Canvas. This will create a Canvas GameObject with a Canvas component.
2. Name the Canvas GameObject “YourTaskName\_Canvas”, replacing YourTaskName with the name of your task.
   1. This is crucial, as our scripts will search for your canvas using this naming structure.
3. Select your Canvas GameObject, then, In the inspector:
   1. Set the Canvas Component’s render mode to “Screen Space – Camera” and drag your scene’s Camera into the “Render Camera” field of the Canvas Component.
   2. Set the Canvas Scaler Component’s UI Scale Mode to “Scale With Screen Size” and set the Reference Resolution to 1920 x 1080.

## Generate Task Namespace

Your task’s namespace (created in **Generate Task Scripts**) is a container for the task’s main classes (TaskDef, BlockDef, TrialDef, StimDef) and contains their fields, properties, and methods. Each class can have a corresponding configuration file, which are loaded and parsed at the **Session Level**.

## Define the TaskDef Class

Define your task’s TaskDef class that inherits from the public class TaskDef from the USE\_Def\_Namespace, which already contains default fields such as TaskName, ExternalStimFolderPath, etc.

## Define the BlockDef Class

Define your task’s BlockDef class. This class inherits from the public class BlockDef from the USE\_Def\_Namespace, which already has its own fields such as BlockCount, TrialDefs, etc. Declare any block variables included in the task’s BlockDef config.

EffortControl\_Namespace.cs

public class EffortControl\_BlockDef : BlockDef

{

   public string BlockName;

   public string ContextName;

   public string TrialId;

   public int NumTrials;

   public int NumClicksLeft;

   public int NumClicksRight;

   public int NumCoinsLeft;

   public int NumCoinsRight;

   public int NumPulsesLeft;

   public int NumPulsesRight;

   public int PulseSizeLeft;

   public int PulseSizeRight;

   public int ClicksPerOutline;

}

For Effort Control, block values are specified in the EffortControl\_BlockDef configuration and the EffortControl\_BlockDef method called GenerateTrialDefsFromBlockDef (shown below) generates the TrialDefs for the blocks. This method is an override method which means it’s our own unique, abstract version of an inherited method from our parent class, BlockDef.

EffortControl\_Namespace.cs

public override void GenerateTrialDefsFromBlockDef()

{

   TrialDefs = new List<EffortControl\_TrialDef>().ConvertAll(x => (TrialDef)x);

// Set to NumTrials in the BlockDef

   for (int iTrial = 0; iTrial < NumTrials; iTrial++)

   {

       EffortControl\_TrialDef td = new EffortControl\_TrialDef();

       td.BlockName = BlockName;

       td.ContextName = ContextName;

       td.TrialId = TrialId;

       td.NumClicksLeft = NumClicksLeft;

       td.NumClicksRight = NumClicksRight;

       td.NumCoinsLeft = NumCoinsLeft;

       td.NumCoinsRight = NumCoinsRight;

       td.NumPulsesLeft = NumPulsesLeft;

       td.NumPulsesRight = NumPulsesRight;

       td.PulseSizeLeft = PulseSizeLeft;

       td.PulseSizeRight = PulseSizeRight;

       td.ClicksPerOutline = ClicksPerOutline;

       TrialDefs.Add(td);

   }

}

## Define the TrialDef Class

This class inherits from the public class TrialDef from the USE\_Def\_Namespace, which has its own fields such as BlockCount, TrialCountInBlock, etc. Declare any block variables included in the task’s TrialDef config. For Effort Control, there is no TrialDef config file. You will notice the same trial and block variables for this reason. The TrialDefs are created from the values provided in the EffortControl\_BlockDef from the GenerateTrialDefsFromBlockDef method shown above.

EffortControl\_Namespace.cs

public class EffortControl\_TrialDef : TrialDef

{

   public string TrialId;

   public string BlockName;

   public string ContextName;

   public int NumClicksLeft;

   public int NumClicksRight;

   public int NumCoinsLeft;

   public int NumCoinsRight;

   public int NumPulsesLeft;

   public int NumPulsesRight;

   public int PulseSizeLeft;

   public int PulseSizeRight;

   public int ClicksPerOutline;

}

## Define the StimDef Class

The Effort Control task does not use stimuli, so there are no additional EffortControl\_StimDef variables defined. If your task uses Quaddle stimuli and needs any additional variables, define them here.

## Setting Up Configuration Folder & Files

See [ConfigurationFileSetup](#_Configuration_Files_Setup) for creating and updating the MUSE/Configs folder. Therein you will create and update the configs folder, the SessionConfig file, the task config folder, and the task’s individual configuration files.

## Setting Up Your Resources Folder & Files

See [ResourcesFolderSetup](#_Setting_Up_the) for creating and updating the MUSE/Resources folder. Therein you will create and update the Resources folder and its Contexts, TaskIcons, and Stimuli sub-folders.

## Defining the Task Level

This class inherits from ControlLevel\_Task\_Template, which includes the following default States: VerifyTask, SetupTask, RunBlock, BlockFeedback, and FinishTask. As shown in the screenshot below, the EffortControl\_TaskLevel gets access to the current BlockDef through the GetCurrentBlockDef method and stores the block locally as currentBlock. This allows us to access and alter the current block configurations data at the **Task Level.** Access to the task’s Trial Level in shown below.

EffortControl\_BlockDef currentBlock => GetCurrentBlockDef<EffortControl\_BlockDef>();

public override void DefineControlLevel()

{

   trialLevel = (EffortControl\_TrialLevel)TrialLevel;

RunBlock.AddInitializationMethod(() =>

   {

trialLevel.ResetBlockVariables();

trialLevel.TokenFBController.SetTotalTokensNum(currentBlock.NumTokenBar);

SetSkyBox();

   });

   BlockFeedback.AddInitializationMethod(() =>

   {

       AddBlockValuesToTaskValues();

       HandleBlockStrings();

}

The Task Level allows you to create a BlockSummaryString that will be updated and displayed on the Experimenter Display during the session, summarizing key data about the current and previous blocks. To accomplish this, create a similar method to Effort Control’s CalculateBlockSummaryString method shown below.

public void CalculateBlockSummaryString()

{

   ClearStrings();

   CurrentBlockString = ("Touches: " + trialLevel.TotalTouches\_Block +

                   "\nReward Pulses: " + trialLevel.RewardPulses\_Block +

                   "\n\nChose Left: " + trialLevel.NumChosenLeft\_Block +

                   "\nChose Right: " + trialLevel.NumChosenRight\_Block +

                   "\n\nChose Higher Reward: " + trialLevel.NumHigherRewardChosen\_Block +

                   "\nChose Lower Reward: " + trialLevel.NumLowerRewardChosen\_Block +

                   "\nChose Same Reward: " + trialLevel.NumSameRewardChosen\_Block +

                   "\n\nChose Higher Effort: " + trialLevel.NumHigherEffortChosen\_Block +

                   "\nChose Lower Effort: " + trialLevel.NumLowerEffortChosen\_Block +

                   "\nChoseSameEffort: " + trialLevel.NumSameEffortChosen\_Block +

                   "\n");

   BlockSummaryString.AppendLine(CurrentBlockString).ToString();

}

Below is an image of the BlockSummaryString on the Experimenter’s Display.

A screenshot of a computer

Description automatically generated

## Defining the Trial Level

The TrialLevel script will contain the majority of the logic related to the task’s function. Every task’s TrialLevel inherits from USE\_ExperimentTemplate\_TrialLevel, which contains the 5 default states: LoadTrialStims, SetupTrial, FinishTrial, Delay, and GazeCalibration. Additionally, tasks can define custom states in the DefineControlLevel method as seen below from the EffortControl\_TrialLevel.

public override void DefineControlLevel()

{

   State InitTrial = new State("InitTrial");

   State ChooseBalloon = new State("ChooseBalloon");

   State CenterSelection = new State("CenterSelection");

   State InflateBalloon = new State("InflateBalloon");

   State Feedback = new State("Feedback");

   State ITI = new State("ITI");

AddActiveStates(new List<State> { InitTrial, ChooseBalloon, CenterSelection, InflateBalloon, Feedback, ITI });

}

If you would like to include an adaptive staircase trial selection procedure in your task, you must implement an override of DefineCustomTrialDefSelection, as seen below in EffortControl\_TrialLevel.

A screen shot of a computer program

Description automatically generated

Additionally, add 1 to the list runningPerformance each time a trial is completed “unsuccessfully,” and add a 0 each time a “successful” trial is completed. This should be done in the task-specific TrialLevel script, wherever you record the correctness or success of a trial.

## Using the State System in a Task

Every state can be further modified with the following methods the pertain to initialization, update, and termination of the state. The lambda expression, which is an anonymous function that allows you to define a delegate inline without explicitly defining a separate method, is utilized by the main methods throughout the state system.

### AddInitializationMethod

AddInitializationMethod adds a method that will be called at every first frame of the state, and is useful for initializing any critical values at the start of the state. Below is an example of the AddInitializationMethod attached to the SetupTrial state to initialize the ConfigUI variables, TaskSummaryString, and TokenFBController variables.

SetupTrial.AddInitializationMethod(() =>

{

   LoadConfigUIVariables();

   if (TrialCount\_InTask != 0)

       currentTask.SetTaskSummaryString();

   if (TokenFBController != null)

       SetTokenVariables();

});

### AddUpdateMethod

AddUpdateMethod adds methods that run every frame of the Unity update cycle that the state is active. This method is ideal for behaviors that may require updating every frame, such as input handling, or animations. Below is an example of the CenterSelection state that is updating the location of the balloon every frame until it is a positioned at the center of the screen, giving an illusion of a continuous movement.

CenterSelection.AddUpdateMethod(() =>

{

   if(Wrapper.transform.position != CenteredPos)

Wrapper.transform.position = Vector3.MoveTowards(Wrapper.transform.position, CenteredPos, CenteringSpeed \* Time.deltaTime);

});

### SpecifyTermination

SpecifyTermination adds a termination criterion, and optional termination methods to a state, and specifies the successor state which will initialize the following frame. Below is an example of the SpecifyTermination method of the CenterSelection state that will continue to the Delay state once the balloon is at the CenteredPos. The method contains an override for a third parameter that is a lambda expression, that can include any methods that occur at the termination of the state.

CenterSelection.SpecifyTermination(() => Wrapper.transform.position == CenteredPos, Delay);

### AddTimer

AddTimer is a quicker way assigning a timer that controls termination of the state: it is equivalent to manually adding a timer using the SpecifyTermination method, but requires less space. Below is an example of the AddTimer method of the ITI state that will continue to the FinishTrial once the itiDuration.value is exceeded. The method contains an override for a third parameter that is a lambda expression, that can include any actions that occur at the termination of the state.

ITI.AddTimer(itiDuration.value, FinishTrial);

### AddDefaultTerminationMethod

AddDefaultTerminationMethod specifies methods that that will occur at the termination of the state, if no specific termination methods have been defined using the lambda overloads of SpecifyTermination or AddTimer. Below is an example of the AddDefaultTerminationMethod method of the Feedback state that contains 2 SpecifyTermination methods as well. Since neither SpecifyTermination method contains a custom termination method, the default termination will run at the end of both SpecifyTermination calls.

Feedback.SpecifyTermination(() => AddTokenInflateAudioPlayed && !AudioFBController.IsPlaying() && !TokenFBController.IsAnimating(), ITI);

Feedback.SpecifyTermination(() => true && Response != 1, ITI);

Feedback.AddDefaultTerminationMethod(() =>

{

   TokenFBController.enabled = false;

   AddTokenInflateAudioPlayed = false;

});

### AddUniversalTerminationMethod

AddUniversalTerminationMethod dictates any actions that occur at the termination of the state, regardless of if a termination is defined elsewhere and works similarly to AddDefaultTerminationMethod, except that its methods are always run after any termination.

# M-USE Modules

## Experimenter Display

A screenshot of a computer

Description automatically generated

### Initializing the Experimenter Display

The Experimenter Display allows real-time feedback of the participant’s performance on a separate monitor. The USE\_ExperimentDisplay namespace contains the public classes ExperimentDisplayController and ExperimenterDisplayPanel and the static class DefaultPanels. The classes allow for the user to customize the types of panels that are included on the secondary display, otherwise default panels are generated. The default panels include TrialInfoPanel, BlockInfoPanel, SessionInfoPanel, LogInfoPanel, and HotKeyPanel, which all inherit from ExperimenterDisplayPanel.

The Experiment Display is created at the Session Level and can be accessed through the ExperimentDisplayController field of the static class SessionValues. At the termination of InitScreen (link), InitializeExperimenterDisplay will either initialize the 5 default panels or custom panels if they are passed as parameters. The ExperimentDisplayController handles all custom panel updates in the Unity Update cycle.

private void CreateExperimenterDisplay()

{

experimenterDisplay = Instantiate(Resources.Load<GameObject>("Default\_ExperimenterDisplay"));

   experimenterDisplay.name = "ExperimenterDisplay";

SessionValues.ExperimenterDisplayController = experimenterDisplay.AddComponent<ExperimenterDisplayController>();

   experimenterDisplay.AddComponent<PreserveObject>();

SessionValues.ExperimenterDisplayController.InitializeExperimenterDisplay(this, experimenterDisplay);

}

### Trial Info Panel

The Trial Info Panel outputs the information pertaining to the ongoing task’s trial data. The USE\_ExperimentTemplate\_Trial contains a TrialSummaryString field that can be modified within the task to display distinct performance metrics. The following method is included in the VisualSearch\_TrialLevel and is called given any updates within the trial.

private void SetTrialSummaryString()

{

   TrialSummaryString = "Selected Object Index: " + SelectedStimIndex +

                        "\nSelected Object Location: " + SelectedStimLocation +

                        "\n" +

                        "\nCorrect Selection: " + CorrectSelection +

                        "\nTouch Duration Error: " + TouchDurationError +

                        "\n" +

                        "\nSearch Duration: " + SearchDuration +

                        "\n" +

                        "\nToken Bar Value: " + TokenFBController.GetTokenBarValue();

}

### Block Info Panel

The Block Info Panel outputs the information pertaining to the ongoing task’s block data. The USE\_ExperimentTemplate\_Task contains a BlockSummaryString and PreviousBlockSummaryString fields that can be modified within the task to display distinct block performance metrics. The panel will automatically display the TaskLevel.BlockCount and TrialLevel.TrialCount\_InBlock. The following method is included in the VisualSearch\_TaskLevel and is called given any updates within the block.

public void SetBlockSummaryString()

{

   ClearStrings();

   BlockSummaryString.AppendLine("\nNum Aborted Trials: " + + vsTL.AbortedTrials\_InBlock +

                                 "\n"+

                                 "\nNum Reward Given: " + vsTL.NumRewardPulses\_InBlock +

                                 "\nNum Token Bar Filled: " + vsTL.NumTokenBarFull\_InBlock

}

### Session Info Panel

The Session Info Panel outputs the information pertaining to every task within the session. The following metrics are automatically calculated and displayed in the panel: totalTrials, totalRewardPulses, sessionDuration, timeFromLastTrialCompletion. The USE\_ExperimentTemplate\_Task contains a CurrentTaskSummaryString that can be modified within the task to display distinct task performance metrics, similarly to the BlockSummaryString. The panel will automatically display the name of the selected tasks within the session if specific summary strings are not generated. The following method is included in the SessionInfoPanel script, and the method UpdateSessionSummaryValues() can be called to update the fields in the script at discrete points within the project.

private void SetSessionSummaryString()

{

   SessionSummaryString.Clear();

   SessionSummaryString.Append(

       "Total Trials: " + totalTrials +

       "\nTotal Reward Pulses: " + totalRewardPulses +

       "\nSession Duration: " + String.Format("{0:0.0}", sessionDuration) + " s" +

       "\nTime From Last Trial Completion: " + String.Format("{0:0.0}",

       timeFromLastTrialCompletion) + " s");

   TaskSummaryString.Clear();

if (TaskLevel != null)

   {

TaskSummaryString.Append("<b>\n\nSelected Configs: </b>" + TaskLevel.CurrentTaskSummaryString);

   }

if (SessionLevel.PreviousTaskSummaryString.Length > 0)

       TaskSummaryString.AppendLine(SessionLevel.PreviousTaskSummaryString.ToString());

}

### Log Info Panel

The Log Info Panel outputs any Debug.LogError messages that occur during a build. The Player.log is saved to the data folder and lists every logged message during the session.

### Hot Key Panel

The Hot Key Panel details every Hot Key that is included within the project and the Config UI Hot Keys that allow keyboard navigation of the Config UI Panel. The HotKeyPanel script contains the public classes HotKey and HotKeyList. Every Hot Key is an instance of HotKey and is added to a single instance of HotKeyList that handles the initialization and checking of all Hot Key Conditions for default and custom Hot Keys. The keyDescription and actionName fields are used to populate the HotkeyPanel so Experimenters can see a list of hotkeys and their corresponding actions.

The following method is included in the HotKeyPanel script and details the addition of a toggleCursor Hot Key that toggles the visibility of the cursor when the C key is pressed.

HotKey toggleCursor = new HotKey

{

   keyDescription = "C",

   actionName = "Cursor Visibility",

   hotKeyCondition = () => InputBroker.GetKeyUp(KeyCode.C),

   hotKeyAction = () =>

   {

       Cursor.visible = !Cursor.visible;

   }

};

HotKeyList.Add(toggleCursor);

### Config UI Panel

The Config UI panel allows for in-session adjustment of task variables. The task must include a ConfigUI configuration file detailing all variables that are configurable in the task, and corresponding variables of type ConfigNumber in the script.

The variables listed in the ConfigUI configuration file are automatically read in and stored inside the ConfigUiVariables variable. Then, you can access, retrieve, and map their values to your scripts corresponding ConfigNumber variables each trial as detailed below. The following method was taken from the VisualSearch\_TrialLevel, and is called during SetupTrial.

private void LoadConfigUIVariables()

{

   //config UI variables

   minObjectTouchDuration = ConfigUiVariables.get<ConfigNumber>("minObjectTouchDuration");

   maxObjectTouchDuration = ConfigUiVariables.get<ConfigNumber>("maxObjectTouchDuration");

   itiDuration = ConfigUiVariables.get<ConfigNumber>("itiDuration");

   searchDisplayDelay = ConfigUiVariables.get<ConfigNumber>("searchDisplayDelay");

   selectObjectDuration = ConfigUiVariables.get<ConfigNumber>("selectObjectDuration");

   fbDuration = ConfigUiVariables.get<ConfigNumber>("fbDuration");

   gratingSquareDuration = ConfigUiVariables.get<ConfigNumber>("gratingSquareDuration");

   tokenRevealDuration = ConfigUiVariables.get<ConfigNumber>("tokenRevealDuration");

   tokenUpdateDuration = ConfigUiVariables.get<ConfigNumber>("tokenUpdateDuration");

   tokenFlashingDuration = ConfigUiVariables.get<ConfigNumber>("tokenFlashingDuration");

   configUIVariablesLoaded = true;

}

### Player View Panel

The Player View panel depicts the ongoing scene from the Player’s monitor onto the Experimenter Display. A RenderTexture CameraMirrorTexture is created according to the screen dimensions and is assigned to the Main Camera’s target texture. CameraMirrorTexture is then applied to the texture of the RawImage of the panel. The CameraMirrorTexture is drawn and updated within the SessionLevel script in the Unity OnGUI system. Text and symbols can be overlayed onto the panel using methods from the PlayerViewPanel script and attaching the objects as a child of MainCameraCopy game object in the scene hierarchy.

public void AssignTaskCamToThePlayerViewPanel ()

{

   CameraMirrorTexture = new RenderTexture(Screen.width, Screen.height, 24);

   CameraMirrorTexture.Create();

   CurrentTask.TaskCam.targetTexture = CameraMirrorTexture;

   mainCameraCopy\_Image.texture = CameraMirrorTexture;

}

public void **OnGUI**()

{

   if (CameraMirrorTexture == null) return;

GUI.DrawTexture(new Rect(0, 0, Screen.width, Screen.height), CameraMirrorTexture);

}

## Feedback Controllers

### Token Feedback

The Token Feedback system can be implemented into any task as a visual representation of reward and task progress. Tokens can be flexibly added and removed given the accuracy of the participant’s selection.

The TokenFBController class is designed to control token feedback, with functionalities related to display, animation, and audio. The script includes parameters for the visual appearance of the tokens and animation phases. It also includes methods for calculating and adjusting the token box, animating tokens, adding or removing tokens, checking if the token bar is full, and adjusting the size of the token bar. Furthermore, it includes methods for setting various properties of the tokens, such as their number, reveal time, update time, and flashing time. Also included is an OnGUI method for rendering and drawing the tokens on the screen. Event codes can be used to manage specific interactions or occurrences related to the token bar. Below is an example from FlexLearning\_TrialLevel.

if (selectedSD.StimTokenRewardMag > 0)

{

   TokenFBController.AddTokens(selectedGO, selectedSD.StimTokenRewardMag);

else

{

   TokenFBController.RemoveTokens(selectedGO, -selectedSD.StimTokenRewardMag);

}

### Slider Feedback

The Slider Feedback system can be implemented into any task as a visual representation of reward and task progress. Slider progress can be flexibly added and removed given the accuracy of the participant’s selection.

The SliderFBController is responsible for managing and controlling slider-based UI feedback within the task. It contains properties to represent the slider and its halo, as well as various methods to initialize, configure, and update these elements. The class uses animations to indicate different phases, including updating and flashing, while also providing audio feedback. Event codes can be used to manage specific interactions or occurrences related to the slider. Other data related to the slider, such as its fullness or the number of times it has been filled, are also handled by this class. Below is an example from MazeGame\_TrialLevel.

if (CorrectSelection)

{

SliderFBController.UpdateSliderValue(selectedGO.GetComponent<Tile>().sliderValueChange);

}

### Halo Feedback

The Halo Feedback system can be implemented into any task as a visual representation of performance accuracy. By default, a “correct” response will display a yellow halo around the selected object while an “incorrect” response will display a gray halo around the selected object.

The HaloFBController class manages the creation, configuration, and destruction of positive and negative halo feedback objects. These objects are instantiated based on provided prefabs and are typically used to visually highlight a game object in 3D or 2D space. This class also provides methods to customize the properties of these halos such as their size, color, and intensity. If halo feedback is already visible when a new one is being created, the old one will be destroyed unless the LeaveFBOn flag is set to true. The state of the halo (none, positive, negative) is logged in the frame data for tracking purposes. Below is an example for VisualSearch\_TrialLevel.

if (CorrectSelection)

   HaloFBController.ShowPositive(selectedGO, depth);

else

   HaloFBController.ShowNegative(selectedGO, depth);

### Touch Duration Feedback

The Touch Duration Feedback system can be implemented into any task as a visual representation of performance. The system handles selections that are “too short,” “too long,” or “dragged.” This system is important for discerning intentional selections and training specific selection durations.

The TouchFBController is a class in Unity that handles the visual feedback for touch interactions, particularly touch errors. It provides touch feedback by instantiating game objects from three different prefabs, HeldTooLong\_Prefab, HeldTooShort\_Prefab, and MovedTooFar\_Prefab, which correspond to the three types of touch errors it can handle. It also tracks the count of each error type in a dictionary Error\_Dict. The class provides methods to enable and disable the touch feedback and manage the display of the feedback objects. The touch feedback is displayed on a canvas TaskCanvas, and its duration and size can be set by the user. EnableTouchFeedback will implement the touch feedback into the task in response to the provided MinDuration and MaxDuration of the Selection Handler. Below is an example from VisualSearch\_TrialLevel.

ShotgunHandler.MinDuration = minObjectTouchDuration.value;

ShotgunHandler.MaxDuration = maxObjectTouchDuration.value;

TouchFBController.EnableTouchFeedback(ShotgunHandler, TouchFeedbackDuration, TouchFeedbackScale, VS\_CanvasGO);

### Audio Feedback

The Audio Feedback system can be implemented into any task as an aural representation of performance. By default, a “correct” response will play a brief, high-pitched tone while an “incorrect” response will play a brief, low-pitched tone. Completion of either the Token Bar or Slider Bar will result in a tri-tone sequence to indicate reward delivery, in the case of a connected Reward Pump.

The AudioFBController class is responsible for managing and playing audio clips based on events within the session. The key methods in this class initialize the audio source and sets up a dictionary to store audio clips, locate an active audio source within the game objects, retrieve and play an audio clip by its name, and add additional audio clips at runtime. The Update() method is also used to check the status of the audio source in each frame and updates the name of the currently playing clip if no clip is playing for the frame data. Below is an example from TokenFBController.

if (tokensChange < 0)

   audioFBController.Play("NegativeShow");

else

   audioFBController.Play("PositiveShow");

## UI Classes

### HumanStartPanel

A blue and yellow text with a play button

Description automatically generatedThe HumanStartPanel class is responsible for generating and displaying the starting screen for the human version of each task. It provides human subjects with added flexibility and navigation, while creating consistency across the tasks. The UI panel is comprised of the start button, the task name, an instructions button, and an end task button.

The SessionValues class contains one static instance of HumanStartPanel, and the USE\_ExperimentTemplate\_Task script automatically calls the CreateHumanStartPanel() method to setup the panel at the start of each task.

For new tasks, add your task’s name (separated by a space) and instructions to the HumanStartPanel dictionaries during the DefineControlLevel() method of your custom task level.

SessionValues.HumanStartPanel.AddTaskDisplayName(TaskName, “Continuous Recognition”);

SessionValues.HumanStartPanel.AddTaskInstructions(TaskName, “Select an object you haven’t chosen!”);

In the AddControlLevel\_InitializationMethod() method of your task’s custom trial level script, you can get a reference to the start button and set the states for which the HumanStartPanel will be visible.

If(SessionValues.SessionDef.IsHuman)

{

StartButton = SessionValues.HumanStartPanel.StartButtonGO;

SessionValues.HumanStartPanel.SetVisibilityOnOffStates(InitTrial, InitTrial);

}

### USE\_StartButton

The USE\_StartButton class is responsible for generating the Start Button for the non-human version of each task. The SessionValues class contains one static instance of USE\_StartButton and the component is added automatically at the session level.

In the AddControlLevel\_InitializationMethod() method of your task’s custom trial level script, you can create the start button, obtain a reference to it, and set the states for which it’s visible.

If(!SessionValues.SessionDef.IsHuman)

{

StartButton = SessionValues.USE\_StartButton.CreateStartButton(CR\_CanvasGO.GetComponent<Canvas>(), CurrentTask.StartButtonPosition, CurrentTask.StartButtonScale);

SessionValues.HumanStartPanel.SetVisibilityOnOffStates(InitTrial, InitTrial);

}

## Eye Tracking

### Setting Up an Eye Tracker

**M-USE** manages data received from eye tracking devices, allowing for automatic calibration before a session begins and flexible toggling of in-task calibration to allow for updates in the quality of the calibration throughout the session. If storing data, gaze data is automatically written in the Session data folder (see Data Management below).

### TobiiEyeTrackerController

The TobiiEyeTrackerController manages the methods and properties derived from the **Tobii** eye tracking system. This class inherits from EyeTrackerController\_Base and serves as a singleton instance, meaning only one instance of this class is intended to exist in a project. Within this class, various fields like iEyeTracker, EyeTracker, and ScreenBasedCalibration represent different components of the **Tobii** eye tracking system.

The mostRecentGazeSample object, which holds the most recent gaze data. The class also includes an Update function that runs every frame, continuously checking and setting up eye tracker components if not already set. The class provides a FindEyeTrackerComponents method which locates and initializes essential components for eye tracking, such as the main eye tracker, the display area, and gaze data subscription.

The TobiiEyeTrackerController is set as a field of the static class SessionValues at the SessionLevel and can be accessed throughout the project.

### TobiiGazeDataSubscription

The TobiiGazeDataSubscription class, manages eye tracking data. Its main function is to enqueue incoming gaze data in a thread-safe manner, then process that data on the main Unity thread for safe use within the Unity engine. This data processing is done in the HandleGazeData method, which captures specific data points from both the left and right eyes such as pupil validity, gaze origin, gaze point, and pupil diameter. All processed gaze data is stored in the mostRecentGazeSample object, ready to be used in Unity's main thread. PumpGazeData must be called during an Update cycle to allow for continual appending of the gaze data and clearing of the accumulated gaze data queue.

public void PumpGazeData()

{

   var next = GetNextGazeData();

   while (next != null)

   {

       HandleGazeData(next);

       next = GetNextGazeData();

   }

}

### Setting Up a Gaze Selection Handler

In order to utilize gaze as the mode of selection, assign the selection handler to be either GazeSelection, GazeShotgun, or a custom selection handler that utilizes gaze position as its input positon.

SelectionHandler = SessionValues.SelectionTracker .SetupSelectionHandler("trial", "GazeSelection", SessionValues.GazeTracker, InitTrial, SearchDisplay);

## Sync Box Controller

The SyncBoxController script controls the operation of the ACCL lab’s custom SyncBox hardware, for timekeeping and I/O during experiments. There are four main methods: SendCommand, which adds a string command to the SerialPortController’s buffer of commands to be sent, SendRewardPulses, SendSonication, and SendCameraSyncPulses, all of which use the SendCommand method to send specific commands associated with different experimental hardware. The EventCodeController is a separate module that is concerned specifically with sending event codes, which also uses SendCommand to control this.

### Setting Up a Sync Box

The SyncBox connects via USB. Experimenters will have to determine its address,

## Event Code Controller

Details need to be added

### Setting Up Event Codes

Details need to be added

## Serial Port Controller

Details need to be added

### Setting Up a Serial Port

Details need to be added

## Input Broker

Details need to be added

### Using the Input Broker

Details need to be added

## Input Tracker

Details need to be added

### Using an Input Tracker

### Mouse Tracker

### Gaze Tracker

### Joystick Tracker

## Selection Handler

Details need to be added

### Creating A Selection Handler

Details need to be added

### Using the Selection Handler

Details need to be added

# Resource Management

## Stimuli Management

### Stim Groups

The StimGroup class allows for client code, most commonly TrialLevel scripts, to arrange and manipulate groups of stimuli as a single unit. This class centers around a list of StimDef objects which constitute the StimGroup.

The method SetVisibilityOnOffStates toggles the visibility of all stimuli in a StimGroup based on the program’s state. The StimGroup appears at whichever state is specified by setActiveOnInit and disappears at the state specified by setInactiveOnTerm.

tStim = new StimGroup("SearchStimuli", group, CurrentTrialDef.TrialStimIndices);

tStim.SetVisibilityOnOffStates(GetStateFromName("SearchDisplay"), GetStateFromName("ITI"));

Similarly, ToggleVisibility allows the user to manually switch the StimGroup’s visibility at discrete points in the script.

StimGroup also contains methods to add and remove StimDefs from the StimGroups on which they are called. Additionally, the SetLocations method allows the user to pass a group of vectors to define the locations of each stimulus in the StimGroup.

### Stim Defs

The StimDef class defines a M-USE stimulus, tracking its associated files, configurations, and behavior. It contains several overloaded constructors, plus methods that facilitate file loading, manage StimGroups, and control stimulus activation.

Fields such as StimPath and PrefabPath are used to handle the loading of files into a StimDef object. Others, like StimIndex, are used to identify the stimulus in a task or within a StimGroup.

The StimDef class also contains several methods handling functionalities related to stimuli loading, positioning, scaling, and managing. Load is a large method that handles the loading of different types of stimuli (2D images, 3D models, and prefabs) based on the results of checking various conditions. It ensures that the appropriate loading method is used (whether that be LoadExternalStimFromFile, LoadExternalStimFromServer, or LoadPrefabFromResources) and provides a callback mechanism to notify the user when the stimulus is loaded.

### Using Stimuli in a Task

To include stimuli in a task, firstly include the file names, indices, and prefab paths (if applicable) of pertinent stimuli in the task’s StimDef config. Then specify the desired stimulus indices and locations in the BlockDef or TrialDef config. To use a stimulus in task scripts, create a child StimDef class in the task namespace (see example below), including any additional task-specific fields or methods to be added onto the parent class. Instantiate this child class in the TrialLevel and use it as desired.

public class WhatWhenWhere\_StimDef : StimDef

{

   //relates to variables to evaluate stimuli

   public bool IsCurrentTarget;

   public bool IsDistractor;

}

You may also instantiate a StimGroup in the TrialLevel as necessary, allowing for easy control over a group of stimuli. In the example below from the WorkingMemory task, searchStims and TrialStims, two previously instantiated StimGroups, allow for easy stimulus management at the trial level.

sampleStim = new StimGroup("TargetStim", GetStateFromName("DisplaySample"), GetStateFromName("DisplaySample"));

for (int iStim = 0; iStim < CurrentTrialDef.SearchStimIndices.Length; iStim++)

{

   WorkingMemory\_StimDef sd = (WorkingMemory\_StimDef)searchStims.stimDefs[iStim];

   sd.StimTokenRewardMag = chooseReward(CurrentTrialDef.SearchStimTokenReward[iStim]);

if (sd.StimTokenRewardMag > 0)

   {

WorkingMemory\_StimDef newTarg = sd.CopyStimDef<WorkingMemory\_StimDef>() as WorkingMemory\_StimDef;

       sampleStim.AddStims(newTarg);

       newTarg.IsTarget = true;

       sd.IsTarget = true; //sets the isTarget value to true in the SearchStim Group

   }

   else sd.IsTarget = false;

}

### Generating Stimuli in Blender

Refer to the Stimulus Generation Manual.

## Context Management

### Generating Contexts

We systematically generated thousands of context images for our task’s backgrounds using the Dalle2 AUI system, which creates realistic images from a description in natural language. This section details the steps we took to accomplish this.

### Preparing Distinct Context Conditions

Create context categories and corresponding sub-categories and adjectives, as shown below.A table with black text

Description automatically generated

A blue and white rectangular box with black text

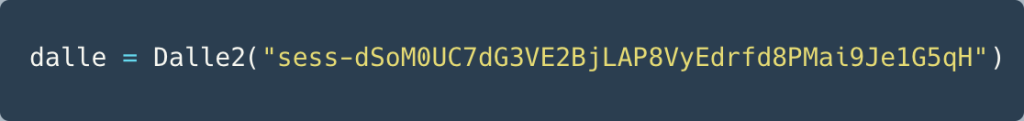
Description automatically generatedSelect artistic styles to apply to each category. Create a numbering system for categories, sub-categories, adjectives, and art styles, giving each a unique identifier.

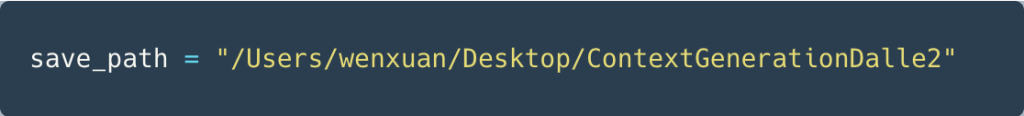
A screenshot of a computer screen

Description automatically generated

Structure the data in excel to contain a row for each category/sub-category, creating individual combinations that are the basis for the Python loops. Use excel formulas (V-Lookups) on the structured data to grab each categories corresponding ID and place it in a column next to the category name. This will allow the python loops to generate the contexts based on their names, and then save the files based on their corresponding IDs.

### Interfacing with Dalle2 in Python

Convert your excel workbook into a CSV file. Go to <https://github.com/ezzcodeezzlife/dalle2-in-python> and follow the setup directions, which will walk you through how to obtain your Bearer Token. Once obtained, add your bearer key to your Python script, as follows:

Add a string variable with the directory path of where to save the contexts:

Write a Python method that reads in the converted CSV file, loops through the string combinations, generating a Dalle2 image for each string, and saving the strings in the directory specified above. The full python script is shown below.

# code for generating images with categories and subjectives

import csv

import requests

import os

total\_number\_generated = 0

save\_path = "/Users/wenxuan/Desktop/ContextGeneratorDalle2"

with open('/Users/wenxuan/Desktop/ContextGeneratorDalle2/context\_number.csv', newline='') as csvfile:

spamreader = csv.reader(csvfile, delimiter=',', quotechar='\"')

for row in spamreader:

print(', '.join(row))

# row[even] = category (from 0) & adjectives, row[odd] = number id (from 1)

for i in range(1,5): # 5 adjectives

num\_image\_generate = 4

for k in range(0, 2): # 2 artstyle

generations = dalle.generate\_amount(row[0].replace("\"", "")+ " " + row[2].replace("\"", "")+ " " + row[2 + i\*2].replace("\"", "")+ " " + row[14 + k \*2] + " "+ "Texture", num\_image\_generate)

total\_number\_generated = total\_number\_generated +1

print("Total number generated " + str(total\_number\_generated))

# save image in url

for j in range(0,num\_image\_generate):

try:

image\_url = generations[0][j]['generation']['image\_path']

img\_data = requests.get(image\_url).content

save\_path = "/Users/wenxuan/Desktop/ContextGeneratorDalle2/" + row[0] +"/"+ row[2] +"/"+ row[2 + i\*2] +"/" + row[14 + k \*2] +"/"

print(save\_path)

try:

os.makedirs(save\_path, exist\_ok = True)

except OSError as error:

print(error)

name\_of\_file = row[1].replace("\"", "") +'\_'+ row[3].replace("\"", "") + "\_" +row[3 + i\*2].replace("\"", "")+"\_"+ row[15 + k \*2] + "\_"+str(j+1)+'.jpg'

completeName = os.path.join(save\_path, name\_of\_file)

with open(completeName, 'wb') as handler:

handler.write(img\_data)

except:

print("Error!!!")

pass

Note: We save each context with a file name matching the numbering system used in the excel file above. This allows for easy categorization and retrieval of specific contexts.

### Using Contexts in a Task

Depending on the task, you may want to set the task’s Skybox (background) on a block basis, or on a trial basis. The EffortControl task sets the context background at the start of each block. To accomplish this, the EffortControl\_BlockDef class has a field for ContextName, and each block in the EffortControl\_BlockDef\_array.txt block configuration file specifies a value for the ContextName. Then, in the EffortControl\_TaskLevel script, at the start of each block, the current block’s ContextName is used to create the Skybox. You pass the file path to the HandleSkyBox coroutine, which loads the texture from the file location and creates the skybox.

RunBlock.AddInitializationMethod(() =>

{

   trialLevel.ResetBlockVariables();

   currentBlock.ContextName = currentBlock.ContextName.Trim();

   SetSkyBox(currentBlock.ContextName, TaskCam.gameObject.GetComponent<Skybox>());

});

# Data Management

This USE\_ExperimentTemplate\_Data is responsible for managing data within a session, by defining various data-related classes and structures that help in collecting, organizing, and storing data generated during an experiment. It includes classes for handling session data, block data, trial data, frame data, gaze data, and serial communication data.

## Session Data

The SessionData class is designed to collect and store essential information about the experiment session. It includes data such as the subject's ID, age, and session timestamp. This class serves as a central repository for session-related data and ensures that this data is properly recorded and organized.

## Block Data

### The BlockData class is responsible for managing data related to experiment blocks. It records information such as the subject's ID, age, session timestamp, the name of the active task, block count, and additional statistics like the number of reward pulses and aborted trials within a block. This data is crucial for tracking the progress of the experiment at the block level.

## Trial Data

## The TrialData is a class dedicated to collecting and storing data associated with individual experiment trials. It captures data such as the subject's ID, age, session timestamp, task name, block count, trial count within a task, and any abort codes associated with trials. This class helps in tracking the specifics of each trial conducted during the experiment.

## Frame Data

The FrameData class focuses on recording frame-related data during the experiment. It collects information like the subject's ID, age, session timestamp, task name, block count, trial count within a task, the current state of the trial, frame count, and the start time of each frame in Unity. This data is useful for analyzing the timing and sequencing of events during the experiment.

## Gaze Data

## The GazeData is responsible for capturing gaze-related data throughout the experiment. It records subject-related information such as ID and age, session timestamp, task name, block count, trial count within a task, the current trial state, frame count, and frame start time in Unity. Additionally, it collects gaze-specific metrics such as pupil diameter and gaze point coordinates. This data is valuable for analyzing subjects' eye movements and visual attention during the experiment.

## Serial Data

## The SerialSentData and SerialRecvData are designed to manage data related to serial communication. The SerialSentData stores data sent through the serial port, while the SerialRecvData records data received via a serial port. These classes enable the recording and analysis of data exchanged between the M-USE software and external hardware or devices via serial communication.

## Summary Data

## The TaskSummaryData class is responsible for summarizing and storing experiment data. It manages the creation of folders for data storage, handles data storage options (local or server), and facilitates the addition of task run data to summary files. This class ensures that experiment data is organized, stored, and retrievable for further analysis or reporting purposes.

## Server Management

### ServerManager Class

The ServerManager static class is responsible for all server-related operations including reading configuration files from an external web server and writing session data files to an external web server. It facilitates communication between the Unity game application and server-side PHP scripts through the use of HTTP requests.

We’ve setup an external web server and placed several PHP scripts on the server to handle the incoming HTTP requests. The ServerManager class methods utilize the UnityWebRequest API to send HTTP requests to the external server, and receive responses asynchronously. The following server interactions are handled by the ServerManager class methods: retrieving files, creating files, appending to files, creating folders, copying folders, and loading textures.

### Server Setup

Setup an external web server that handles HTTP requests, and setup a username and password for SSH access to the server.

SSH into the server by opening a terminal, typing in “ssh UserName@IpAddress”, replacing “UserName” with your account username, and “IpAddress” with the IP Address of your server (Ex: 129.50.200.8). Then, hit enter.

Next, enter the password associated with your account and hit enter.

Now that you’ve accessed the server, use the “cd” command to navigate to root main folder location where you will place the PHP scripts, CONFIG folder, DATA folder, and Resources folder.

Placing the PHP Scripts on the server:

The PHP scripts are contained within the project at M-USE/USE\_CORE/Assets/\_Scripts/GeneralScripts/PHP\_Scripts and you will need to copy them over to your server. Outside Unity, go into your project folder and open a terminal at the location of the PHP\_Scripts folder. Then, type in “scp -r \* UserName@IpAddress:/path/to/root/folder/”, replacing “UserName” with your account username, replacing “IpAddress” with your server’s IP Address, and replacing “/path/to/root/folder/” with the path to your main folder. Then hit enter. This will copy the PHP scripts to the folder location on your server. Ensure the scripts were successfully added to the server, as they play a crucial role in the server communications happening throughout the session.

Creating the CONFIGS folder on the server:

Next, inside your root folder location on the server, use the “mkdir CONFIGS” command to create a folder called “CONFIGS”. Herein you will place any custom Session Config folders you’ve generated for your sessions (See [ConfigSetup](#_Configuration_File_Setup) for setting up config files).

To add Config folders to the CONFIG Folder on your server, Open a terminal at the folder location where the session config folders are that you wish to push. Then, in the terminal, type in “scp -r \* UserName@IpAddress:/path/to/ConfigsFolder/“, replacing “UserName with your account username, “IpAddress” with the IP Address of your server, and “/path/to/ConfigsFolder” with the path to your CONFIGS folder. Then, hit enter. This will copy the session config folders to the CONFIGS folder on the server.

Creating the DATA folder on the server:

Next, inside the root folder location on your machine, use the “mkdir DATA” command to create a folder called “DATA” on the server, wherein the data from each session will be written. You can name it whatever you wish, just make sure you remember the name of the folder, as you will input its name into the Data Folder input field on the Init Screen at the start of the Session.

To copy data to your local machine from the DATA folder on the server, open a terminal at the folder location where you want the data to be copied to. Then, in the terminal, type in “scp -r UserName@IpAddress:/path/to/DataFolder .” and hit enter. This will copy the contents of the server’s DATA folder into the folder location where you opened the terminal.

Creating the Resources folder on the server:

Next, inside your root folder location on your server, use the “mkdir Resources” command to create a folder called “Resources” on your server. Inside it, enter the following commands to create the Contexts, TaskIcons, and Stimuli folders, respectively: “mkdir Contexts”, “mkdir TaskIcons”, “mkdir Stimuli”. (See [ResourcesFolderSetup](#_Resources_Folder) for setting up your resources folder).

Note: the PHP Scripts, CONFIGS folder, Data folder, and Resources folder all need to be at the same directory level on the server.

### Connecting to Server During a Session

A screenshot of a computer

Description automatically generatedIn Unity, when you start the session, the following Initialization Screen will appear. Herein you will Input critical information for your session and connect to your server.

Complete the Config Type input field. Select the Server toggle option if you wish to use a session config folder housed on your server (that you’ve placed inside the CONFIGS folder). Complete the Data Storage input field by selecting the box next to the Server toggle option if you wish to have the data from your session written to your server. If you selected the Server toggle option for either of the Config Type or Data Storage fields, next complete the Server URL field, inputting the URL path to your external web server (e.g., http://m-use.psy.vanderbilt.edu:8080).

Click Connect, so that the connection to the server is tested. Ensure you’ve placed the PHP scripts on your server as mentioned in Server Setup & Folder Structure. If the test is unsuccessful, you will hear an error beep and the Connect Button’s color will turn to red. If the test is successful, the Config Folder dropdown menu will populate with the name of the session config folders inside your root CONFIGS folder. In addition, the Connect button’s color will change to green and its text will change to Connected, indicating the connection to the server is valid.

If your connection was validated, and you chose the Server option for the Config Type field, select the Session Config Folder you wish to use for the session, from the Config Folder dropdown. If your connection was validated, and you chose the Server option for the Data Storage field, complete the Data Folder field, inputting the name of the root data folder you created in the **Server Setup & Folder Structure** (e.g., DATA). Click the Confirm button to submit your information.

Config Loading

After clicking Confirm , the configuration files are fetched and read in from the selected Session Config folder on the server. The Session Config File is read in first from the server. Next, the individual task configuration files for each task listed in the Session Config’s TaskMappings variable are read in and verified.

Data Writing

After clicking Confirm , we create a unique Session Data Folder on the server, inside the Root Data Folder, to store the session’s data.

The following data is written to the server’s Session Data Folder throughout the session:

1. **Session** **Data**
2. Task Data (**Block Data, Trial Data, Frame Data**)
3. **Summary Data** for each task
4. Task Selection **Frame Data**
5. Session Settings
   1. A copy of the Session Config Folder used for the session.
6. Log File

* A file containing all log messages from the session.

# Troubleshooting

## Debugging Approaches

### Using the Log File

Effective debugging is essential to ensure smooth and error-free game play. One powerful debugging tool at your disposal is the Player.log file, which is automatically generated and saved at a location you specify during the Initialization Screen. This log contains a comprehensive record of all debug logs, warnings, and errors that occur during the session. Our state system has been thoughtfully designed to include automatic debug statements, which are logged at both the initialization and termination of each state. These built in debug statements are instrumental in swiftly pinpointing the exact state where any issue may have arisen. Furthermore, any additional manual debugging statements that you incorporate into your scripts will also be captured in the Player log.

### Debug Statements

In conjunction with the automatic debug statements integrated into our state system, we strongly recommend incorporating your own custom debug statements into your scripts. These statements serve as invaluable breadcrumbs throughout the development process, aiding in the identification and resolution of issues as you refine your task. By strategically placing debug statements at key points in your code, you gain real-time insights into variables, conditions, and function executions. This proactive approach to debugging not only accelerates the troubleshooting process but also enhances the overall quality and reliability of the task.

### Using the Unity Profiler

The Unity Profiler is an invaluable tool for developers, as it provides deep insight into the performance of their games, offering a visual representation of how resources like CPU, GPU, memory, and rendering are being utilized during runtime. This data helps identify bottlenecks, memory leaks, and other performance issues that may be affecting the game’s frame rate or causing unexpected behavior.

*Common Errors:*

**Unity Plastic**

A screenshot of a computer

Description automatically generated

Solution: Delete manifest.json in USE-CORE/packages. Reimport any missing packages as a result and exit and reload the Unity window.

**Newtonsoft**

A screenshot of a computer

Description automatically generated

Solution: Import the Newtonsoft package through the menu bar at the top of the Unity Window. Navigate to Window > Package Manager, and in the top-left, select the plus sign and “Add package from git url” and enter the following:  
[com.unity.nuget.newtonsoft-json@3.0](mailto:com.unity.nuget.newtonsoft-json@3.0)

**StandaloneFileBrowser**

A screenshot of a computer

Description automatically generated

Unzip the StandaloneFileBrowser into the LocateFile folder of the USE-CORE project.

A screenshot of a computer

Description automatically generated