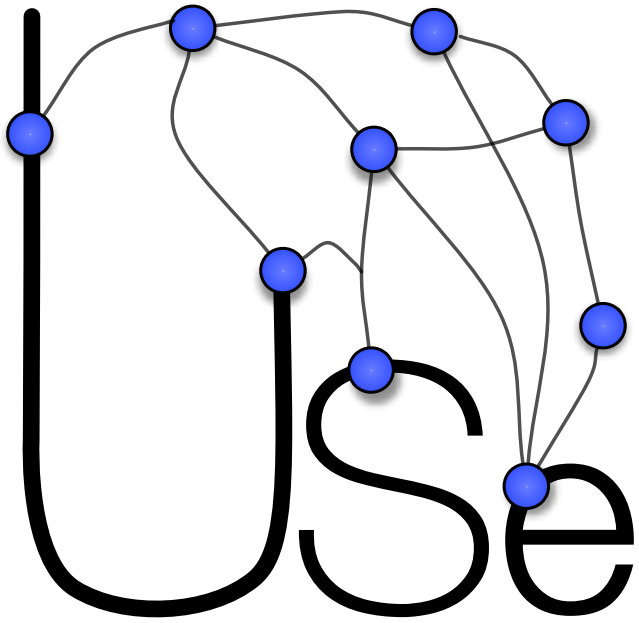


Using USE – Tutorials

Marcus Watson

Table of Contents

1 Tutorial Overview

1.1 Introduction

Each of the tutorials presented here is intended to introduce important aspects of the *Unified Suite for Experiments* (*USE*). They are intended to be worked through in sequence, as each step builds on the previous one. By the end, you should be capable of developing a fully-fledged dynamic experiment of your own in USE.

1.2 Before you start – Unity tutorials

These tutorials assume a basic knowledge of the Unity3D game engine and editor. If you do not have much experience with these, you should go through several of the excellent Unity-specific tutorials at https://unity3d.com/learn/tutorials prior to working through our examples. In particular, the Interactive Tutorials, Roll-a-Ball, Space Shooter, and Raycast tutorials are recommended. This will be at least a full day's work, depending on your level of comfort with code.

1.3 Before you start – Object-oriented programming

C# is a highly *object-oriented* language, and both Unity3D and USE use *object-oriented programming* (OOP) extensively. It is not required that you have a complete understanding of OOP principles to develop an experiment, but it is worthwhile reading some introductions to the basic concepts if you do not have an OOP background (cf. https://en.wikipedia.org/wiki/Object-oriented_programming, https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep).

These tutorials will repeatedly refer to "objects" in Unity3D scripts, which are not necessarily anything resembling objects in the virtual world of an experiment (though they may be), but rather are simply objects in an OOP sense. For example, a single epoch of a trial, such as the feedback period, will be an object of class State, and all objects of type State have all the properties of Unity's Monobehavior class. That is to say *a given trial state is an instantiation of the State class, which inherits from Unity's Monobehavior class*. If you do not understand the general meaning of the italicized phrase, it may be difficult to understand the rationale behind some of the code needed to run experiments in USE, and it may be worth reading up more on OOP.

1.4 Before you start – Download Unity3D

Unity is available for download at https://store.unity.com. Use the Personal plan, which is free, unless your lab makes over $100,000 (US) in sales a year (in which case, why on earth are you still coding experiments?). The standard install package (Unity Editor and Visual Studio) is sufficient for most purposes, along with whichever additional platform's build support you require. There is no need to download the documentation, unless you plan to work without a connection to the internet, as Unity's documentation is all available online and updated regularly.

1.5 Before you start – Download USE and tutorial files

To complete these tutorials, you will need the USE_Core package, and the USE_Tutorial package, available at ***** and *****, respectively.

2 Tutorial 1 – Introduction to Control Levels and States

2.1 Description

In this tutorial we will create a very simple experimental task. There is no interaction with other experimental hardware, no writing of data files, and not even blocks of trials – this is as simple as a task can get.

Each trial will consist of four epochs, which will be defined by States in the Control Level. These include:

- *Stimulus presentation* – a stimulus appears, followed by a brief waiting period.
- *Response* – a response cue appears, and the participant must use the mouse to click on the stimulus before a timer runs out.
- *Feedback* – the response cue disappears, and visual feedback is given, indicating that the participant clicked on the stimulus, clicked on something else, or ran out of time.
- *Inter-trial interval* – the stimulus and feedback disappear, and a brief waiting period ensues before the next trial.

Coding this task introduces several important elements of USE, including:

- creating a Control Level
- adding States to a Control Leveladding Initialization, Update, and Termination methods to States
- using Timers as Termination method shortcuts
- adding multiple Termination methods to a single state
- using the InputBroker to collect mouse clicks and mouse position

As this is the first tutorial, it goes into much more detail about basic tasks in Unity, and gives much more precise instructions, than the later tutorials.

2.2 Experiment Creation

2.2.1 Overview

This tutorial involves importing the USE_Core scripts to a Unity project, creating a new scene for an experiment, and writing a single script that defines a Control Level with four States, one for each of the trial epochs. It also involves creating several objects in Unity's scene hierarchy: empty objects used like folders to keep the hierarchy organized, a simple sphere stimulus, and a 2D canvas containing text and image objects which are used as a response cue and feedback signal, respectively. The States are coded to control the visibility and content of these objects, as well as controlling timing and parsing participant input.

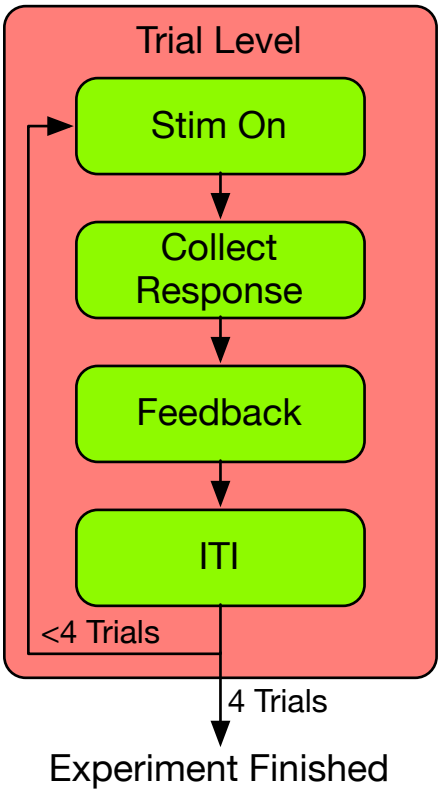


Figure 1. Sketch of the Control Level in Tutorial 1.

2.2.2 Creating the tutorial project, importing USE scripts, changing settings

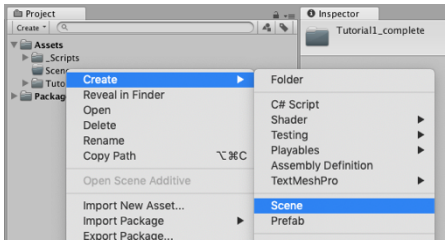
Begin by opening Unity and creating a new Project. Name it *USE_Tutorials*. Place it wherever you find convenient, and keep the Template as 3D. Open it. In the Project Tab, right click on the Assets folder to create a new sub-folder called *_Scripts* (the "_" ensures that it will be at the top of the list of contents of Assets, which is useful).

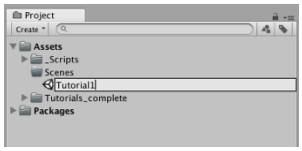
Now add the USE_Core package you downloaded to the _Scripts folder. This can be done either by right-clicking on _Scripts and selecting *Import Package* / *Custom Package* and navigating to the appropriate folder on your hard drive, or simply dragging and dropping the package from the file system. In either case, you will be presented with a dialogue screen showing the package contents. Make sure they are all selected, and click *Import*.

By default, Unity uses older codebases that do not support the full functionality of USE. To change this, go to the Edit Menu, then select Project Settings / Player. In the Inspector, scroll down to the Configuration heading. Change the *Api Compatability Level* to .NET 2.0 and the *Scripting Runtime Version* to .NET 4.x. This will require restarting the editor.

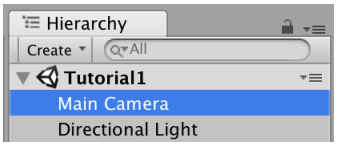
2.2.3 Creating a scene and Control Level script

Next, right-click on the Scenes folder and select Create/Scene. Rename the new scene *Tutorial*.

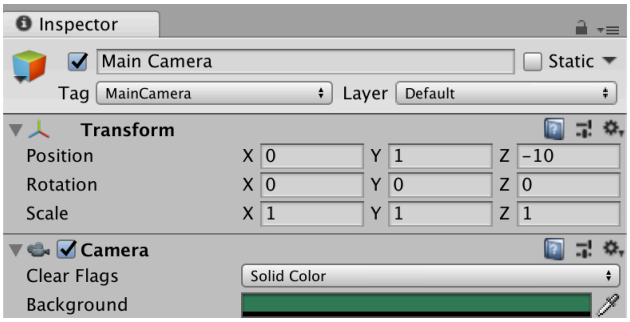




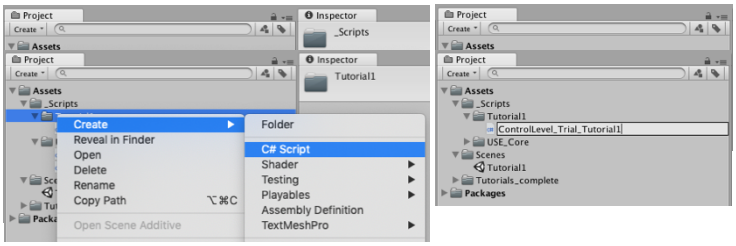
We will give this scene a solid background. Click on the Main Camera object in the scene Hierarchy.



In the Inspector to the right of the window, change the Clear Flags option from Skybox to Solid Color. You can then set the Background color to an appropriate shade (we used RGB values of {50,121,88}).



Then right-click on the _Scripts folder and create a new sub-folder. Rename it *Tutorial*.



Create a new C# script inside the new sub-folder and name it *ControlLevel_Trial.*

2.2.4 Setting up the Control Level

Open the new script. It should look like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ControlLevel_Trial : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

`ControlLevel_Trial` is currently a class which inherits from Unity's `MonoBehavior` class, but this is not the behavior we want. Change the script by including the `USE_States` and `UnityEngine.UI` namespaces, changing `ControlLevel_Trial`'s inheritance to the `ControlLevel` class instead of `MonoBehavior`, and delete the `Start` and `Update` methods. The result should look like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using USE_States;

public class ControlLevel_Trial : ControlLevel {

}
```

You will notice that this script already produces an error in the class declaration: "ControlLevel_Trial does not implement inherited abstract member DefineControlLevel." This is because all Control Levels must include a method called `DefineControlLevel`, which, as the name implies, is used to define the Control Level's operation.

`DefineControlLevel` is an abstract method, which means it requires the `override` keyword in its declaration. Declare it, define the four trial states, and add them to the Control Level:

```
public class ControlLevel_Trial : ControlLevel {

    public override void DefineControlLevel()
    {

        //define States within this Control Level
        State stimOn = new State("StimPres");
        State collectResponse = new State("Response");
        State feedback = new State("Feedback");
        State iti = new State("ITI");
        AddActiveStates(new List<State> { stimOn, collectResponse, feedback, iti });
    }
}
```

Each of the four States is defined with a name string, which will be referenced in debugging and various other types of output. We add them to the Control Level with the `AddActiveStates` command.

Adding states to Control Levels can be done in several ways. For now, we will simply note that it is possible to add them one at a time, not in a list. So instead of the final line in the previous example, we could have written:

```
        AddActiveStates(stimOn);
        AddActiveStates(collectResponse);
        AddActiveStates(feedback);
        AddActiveStates(iti);
```

A Control Level is always in one, and only one, State. By default, the initial State is simply the first one that is added (so in either of the examples above, `stimOn`). This can be overridden with the `SpecifyCurrentState` command, e.g. if we wanted the experiment to start in the feedback State for some reason:

```
        SpecifyCurrentState(feedback);
```

However since we want the experiment to start with `stimOn`, there is no need to do this in the current code.

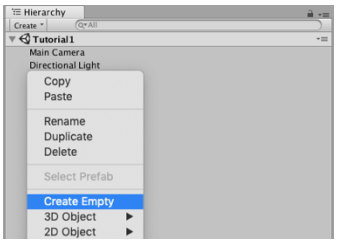
Next we declare variables that represent the three elements of the scene that will change (the stimulus, the response cue, and feedback objects), and the two task variables we need to keep track of (what the response was, and what the current trial number is). These are all public variables, as they will later need to be accessed by methods outside this Control Level, and thus they are declared inside the class definition but before the `DefineControlLevel` method:

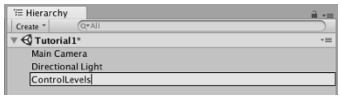
```
public class ControlLevel_Trial : ControlLevel
{
    //scene elements
    public GameObject trialStim;
    public GameObject goCue;
    public GameObject fb;

    //trial variables
    [System.NonSerialized]
    public int trialCount = 1;
    [System.NonSerialized]
    public int response = -1;

    public override void DefineControlLevel()
    {
```

The `trialCount` and `response` variables are declared as `NonSerialized`, which has two effects. First, changes to them will not be saved, and they will revert to their default values after closing the application. Second, they will not appear as a property of the Control Level object in Unity's editor, which is desirable, since we do not need point-and-click access to them.

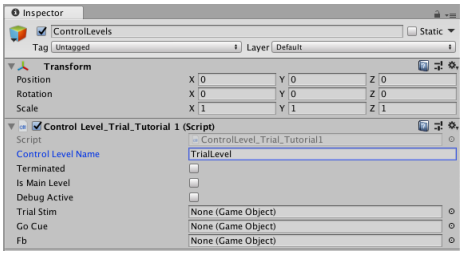
2.2.5 Naming the Control Level, defining stimuli and cues in the Unity Editor

Return to the Unity Editor. In the scene Hierarchy, right click in the empty space to create a new empty object, and name it *ControlLevels*:

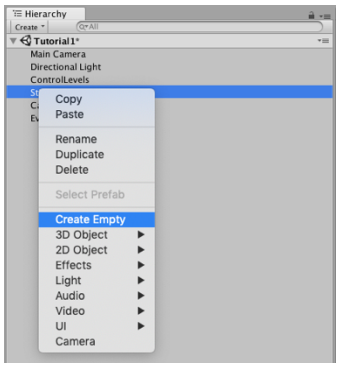


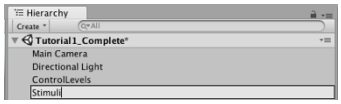


Drag the ControlLevel_Trial script onto the ControlLevels object. After clicking on ControlLevels, you can look in the Inspector and see that the script is now a component of the ControlLevels object. It has a field specifying its name, which is currently empty. Give it the name "TrialLevel":

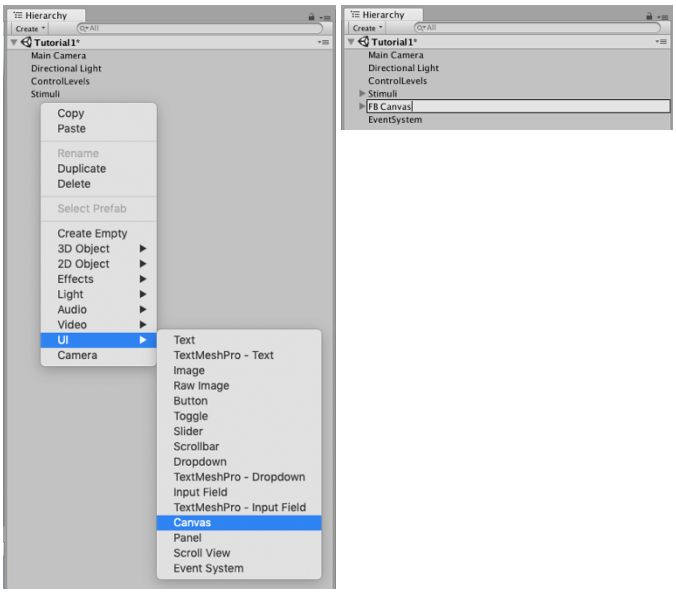


Now we create the three GameObjects needed for this study. This could be done in code, but it is often easier to create simple objects directly in the Unity Editor. In the Hierarchy tab, right-click and create one more empty object. Rename it *Stimuli*.

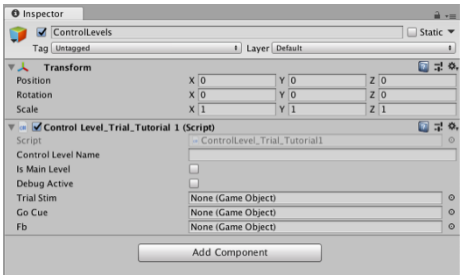




Then create a UI Canvas, and rename it *FB Canvas* (note that this also automatically creates an *EventSystem* object).

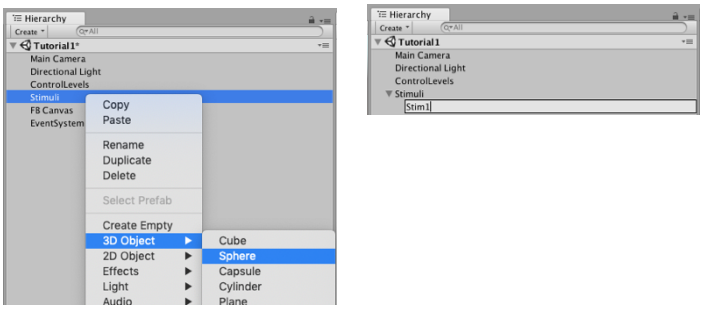


Drag the ControlLevel_Trial script from the Project tab onto the ControlLevels object in the scene hierarchy. Now when you click on the ControlLevels object, you can see an instance of the script has been added as a component in the Inspector. The three public GameObject variables we declared above are fields of this component and are currently empty.

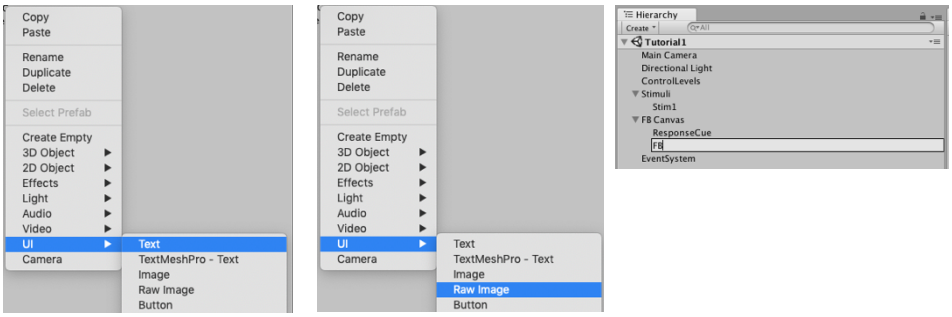


We will now generate the objects to populate them.

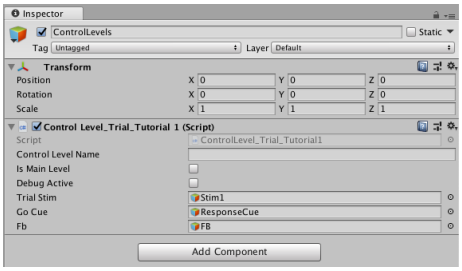
Right click on the *Stimuli* object in the scene hierarchy, and create a new 3D Object/Sphere. Rename it Stim1.



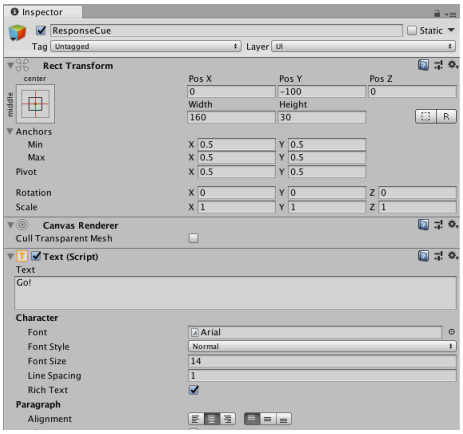
Then right-click on the *FB Canvas*, and create a UI/Text object to be named *ResponseCue*, and a Raw Image to be named *FB*.



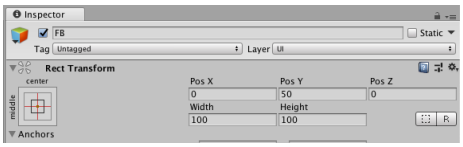
Click on the ControlLevels object again, and drag *Stim1*, *ResponseCue*, and *FB* to the *Trial Stim*, *Go Cue*, and *Fb* fields, respectively, of the *ControlLevel_Trial* component in the Inspector.



Click on the *ResponseCue* object in the scene hierarchy, and in the Inspector edit its Y-position to be -100, its text to read “Go!”, and its alignment to center.



Finally, click on the *FB* object, and edit its Y-position to be 50.



We now have our three GameObjects fully defined. The final step is to set them all to inactive, so they are not visible at the start of the experiment. This is done by unchecking the checkbox at the top left of the inspector for each of the three objects.

Now, we simply need to write the code to run the rest of the task.

2.2.6 The stimOn State

Return to Visual Studio and begin defining the states:

```
        //Define stimOn State
        stimOn.AddStateInitializationMethod(() =>
        {
            trialStim.SetActive(true);
            response = -1;
            Debug.Log("Starting trial " + trialCount);
        });
        stimOn.AddTimer(1f, collectResponse);
```

Here we are defining the initialization and termination of the `stimOn` state. Note that we do not define any update functions, because there is nothing that `stimOn` needs to do on a frame-by-frame basis.

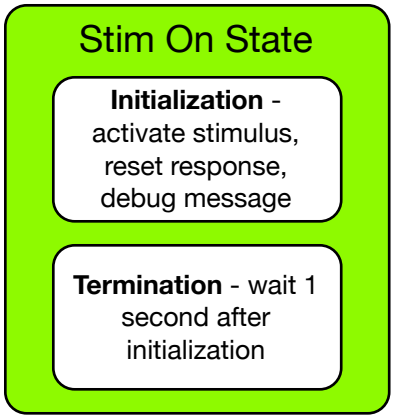


Figure 2. The stim on state.

The `AddStateInitializationMethod` command adds a set of methods to the initialization of the `stimOn` State, which make the stimulus appear, set the value of the response variable to -1, add 1 to the trial count, and print a line of text to the console and log file reporting that the trial has started. The `AddTimer` method makes a special termination condition to the `stimOn` State, such that it will terminate after 1 second, and then pass control to the `collectResponse` State.

There are several important things to consider in this code.

First, what is `()=>`? The *lambda operator* (`()=>`) is commonplace in USE, and passes *references* to functions and variables, rather than the *results* of functions or the *values* of variables. This is a subtle distinction, but an incredibly important one.

Consider the Debug.Log line in that snippet of code. If we isolated it, the `AddStateInitializationMethod` might look like this:

```
        stimOn.AddStateInitializationMethod(() => Debug.Log("Starting trial " + trialCount);
```

Every time the stimOn State initializes, it will run whatever Initialization methods have been added to it. Which in this case is `() => Debug.Log("Starting trial " + trialCount)`. `Debug.Log` outputs a line of text to Unity's console and log files, which consists of the text "Starting trial " and the integer value of `trialCount`. So in every Initialization of the stimOn state, it will find the current value of `trialCount`, and append it to "Starting trial ". Thus after three trials, the console / log file would read:

```
Starting trial 1
Starting trial 2
Starting trial 3
```

But suppose instead we didn't use the lambda operator:

```
        stimOn.AddStateInitializationMethod(Debug.Log("Starting trial " + trialCount);
```

In this case, the value of trialCount used will be the value it had *at the time of adding the initialization method*. It will never be updated, because we have passed the *value* of `trialCount`, not a reference to the variable itself. At the time of adding the initialization method, the value of trialCount is 1, and and so after three trials, the console / log file would read:

```
Starting trial 1
Starting trial 1
Starting trial 1
```

This is clearly not the behavior we want! This is why we use lambda operators (`()=>`) throughout USE, to allow us to pass references to actual functions and variables, instead of the current outputs or values of these functions and variables.

Also, in the previous two examples, we wrote the `AddStateInitializationMethod` slightly differently than in the original code. For all the functions that add methods to USE States, we can either add one line at a time, as in:

```
        stimOn.AddStateInitializationMethod(() => Debug.Log("Starting trial " + trialCount);
```

or we can combine multiple lines, as we did in the original code:

```
        stimOn.AddStateInitializationMethod(() =>
        {
            trialStim.SetActive(true);
            response = -1;
            Debug.Log("Starting trial " + trialCount);
        });
```

There are two basic differences here – when adding multiple lines, they are surrounded with `{}`, and each line terminates with a semicolon. Note that we could also add these multiple lines one at a time, and so the previous code is functionally identical to this:

```
        stimOn.AddStateInitializationMethod(() => trialStim.SetActive(true));
        stimOn.AddStateInitializationMethod(() => response = -1);
        stimOn.AddStateInitializationMethod(() => Debug.Log("Starting trial " + trialCount);
```

It is easier to read and write the combined version, but either is acceptable.

Finally, the `AddTimer` method is a special way of adding a termination specification to a State:

```
        stimOn.AddTimer(1f, collectResponse);
```

`AddTimer` triggers State termination as soon as the duration from the onset time of the first frame in the State to the onset time of the current frame is more than some number of seconds, in this case, 1. Note that decimal values are permitted, but because onset times are only updated once per frame, there is no effective specification of times possible at a granularity of less than one frame duration (16.7 ms on a standard consumer-grade monitor). After termination, `AddTimer` passes control to the State specified in its second argument, in this case `collectResponse` State. We will see the generic way of adding termination specifications when we review this state in the following section.

2.2.7 The collectResponse State

Add the following after the `stimOn` code:

```
        //Define collectResponse State
        collectResponse.AddStateInitializationMethod(() => goCue.SetActive(true));
        collectResponse.AddStateUpdateMethod(() =>
        {
            if (InputBroker.GetMouseButtonDown(0))
            {
                Ray ray = Camera.main.ScreenPointToRay(InputBroker.mousePosition);
                RaycastHit hit;
                if (Physics.Raycast(ray, out hit))
                {
                    if (hit.collider.gameObject == trialStim)
                    {
                        response = 1;
                    }
                    else
                    {
                        response = 0;
                    }
                }
                else
                {
                    response = 2;
                }
            }
        });
        collectResponse.AddTimer(5f, feedback);
        collectResponse.SpecifyStateTermination(() => response > -1, feedback);
        collectResponse.AddStateDefaultTerminationMethod(() => goCue.SetActive(false));
```

The first line makes the response cue visible, just as we made the sphere stimulus visible in `stimOn`. Then we response collection handling to `collectResponse`'s StateUpdate, which you should recall is run once every frame. This requires some more explanation.

First, the `InputBroker` is a component of USE that allows for many different forms of input to be specified. For example, if artificial agents are running a task, it allows their output to be treated the same as more standard ways of interacting with a computer. Thus if the participant is a human using a mouse, `InputBroker.GetMouseButtonDown(0)` checks if the left mouse button is down in precisely the same manner as Unity's native `Input.GetMouseButtonDown(0)`. (In fact, `InputBroker` actually calls the native function.) However if an artificial agent is running the task, `InputBroker.GetMouseButtonDown(0)` will parse the agent's output in a pre-defined way and check if it includes a simulation of a left mouse button press. Other sources of input could be substituted as needed. For the purposes of this tutorial, then, we could substitute Input for InputBroker, but we suggest using InputBroker where possible to allow for easy substitution between different types of participants and response collection equipment.

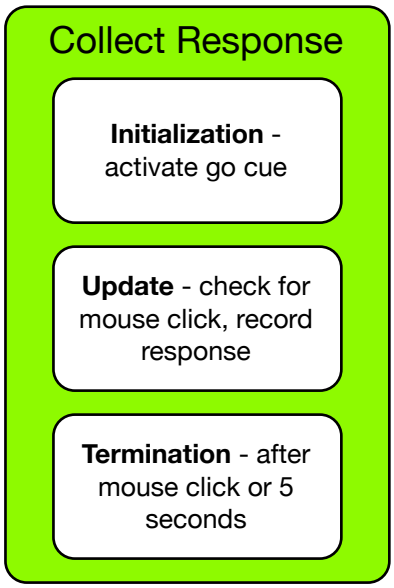


Figure 3. Collect response state.

The rest of the Update function is simple. `Camera.ScreenPointToRay` turns a point on the screen into a `Raycast` emitted perpendicularly from the camera into the worldspace, and so by doing this under the current mouse position, it will return any colliders that appear at this position on the screen. Then, we check to see if the click was on the trial Stimulus. Note that there are two different ways participants could be wrong on a task of this nature: they could click on the wrong object (the first `else` statement), or they could click on an area of the screen with no objects in it (the second `else` statement). Since

this task only has one object, these are identical for the time being, but we will be expanding later tutorials to include multiple objects, and so we allow for this possibility ahead of time.

The `AddTimer` here works the same as in the `stimOn` State. After 5 seconds, control will pass to the `feedback` State. Note, however, that this is not the main thing we want the `collectResponse` State to do – we want it to terminate after a response has been collected, so if a click is made in 1 second, it should end then without waiting the whole five seconds. This is what the following line enforces.

The key concept here is that *States can have multiple ways of handling termination*. Thus the `collectResponse` State will end if 5 seconds have elapsed since the onset of its first frame, or if `Response > -1` (which is to say the user has clicked the left mouse button), whichever happens first.

We previously mentioned that the `AddTimer` method is a special way of specifying a State termination. The standard way of doing this is using the `SpecifyStateTermination` method, as we do here:

```
        collectResponse.SpecifyStateTermination(() => response > -1, feedback);
```

The first argument provides the termination criterion (note it uses a lambda function, otherwise the variable `response` would always have its initial value of -1, and hence this condition would always be false). The second argument, as with the `AddTimer` method, specifies the next State.

Now that we have seen a `SpecifyStateTermination` in action, we can briefly note how the `AddTimer` method actually works. Each State object has a public `StartTimeAbsolute` field, which contains the State's first frame onset time in seconds since the start of the experiment, obtained using Unity's `Time.time` function. The following two lines, then, are exactly equivalent:

```
collectResponse.AddTimer(5f, feedback);

collectResponse.SpecifyStateTermination(() => Time.time – collectResponse.StartTimeAbsolute >= 5f,
feedback);
```

Finally, the last line of the State definition specifies what happens after either of the termination criteria have been reached, namely the response cue is made invisible:

```
        collectResponse.AddStateDefaultTerminationMethod(() => goCue.SetActive(false));
```

Note that as "`AddStateDefaultTerminationMethod`" implies, this line adds a *default* termination method to `collectResponse`, which will be run after any termination criteria has been reached that does not have a specified termination method. Since neither the timer nor the response collection criterion specify their own termination method, the default method will be run. However we could instead have the code:

```
collectResponse.AddTimer(5f, feedback, () => SomeFunction);
collectResponse.SpecifyStateTermination(() => response > -1, feedback, () => SomeOtherFunction);
```

In this case, `SomeFunction` will follow five seconds without a mouse click, and `SomeOtherFunction` will follow a mouse click.

If no default termination method is specified, then the first-specified method is assumed to be the default. Thus the last three lines of this State definition:

```
        collectResponse.AddTimer(5f, feedback);
        collectResponse.SpecifyStateTermination(() => response > -1, feedback);
        collectResponse.AddStateDefaultTerminationMethod(() => goCue.SetActive(false));
```

are functionally identical to:

```
        collectResponse.AddTimer(5f, feedback, () => goCue.SetActive(false));
        collectResponse.SpecifyStateTermination(() => response > -1, feedback);
```

In closing this section, we note that a full specification of a State termination involves 4 things: a termination criterion, a termination method, a successor state, and the specification of the initialization method of the successor state. Just as States can have multiple ways of handling termination, they can also have multiple initialization methods. And just as the first-specified termination method is considered the default if no other method is explicitly defined as the default termination method, a State's first-specified initialization method is considered the default if no other method is explicitly defined as the default initialization method. This is true for all States, in the current task. Further discussion of how to specify complete terminations and initializations will come in a later tutorial.

2.2.8 The feedback and iti states

We finish our State definitions with code that should be fairly simple to read by now:

```
        //Define feedback State
        feedback.AddStateInitializationMethod(() =>
        {
            fb.SetActive(true);
            Color col = Color.white;
            switch (response)
            {
                case -1:
                    col = Color.grey;
                    break;
                case 0:
                    col = Color.red;
                    break;
                case 1:
                    col = Color.green;
                    break;
                case 2:
                    col = Color.black;
                    break;
            }
            fb.GetComponent<RawImage>().color = col;
        });
        feedback.AddTimer(1f, iti, () => fb.SetActive(false));

        //Define iti state
        iti.AddStateInitializationMethod(() => trialStim.SetActive(false));
        iti.AddTimer(2f, stimOn);
```

The `feedback` State starts by making the feedback square visible, and changing its color depending on the response. Note that there are four possible responses: participants might have not clicked in the full five seconds of the collectResponse State (response = -1), or they might have clicked on the *wrong* object, *right* object, or *no* object (response = 0, 1, or 2, respectively). (As previously noted, there is no wrong object at present, but there will be in future tutorials.) After 1 second, the `feedback` State terminates, making the feedback square invisible on its final frame, and passes control to the `iti` state.

The `iti` state simply makes the sphere invisible, and then waits two seconds before iterating the trial counter and passing control back to the `stimOn` State for the subsequent trial. We now have a complete definition of all States in the Control Level, and a nearly complete task!

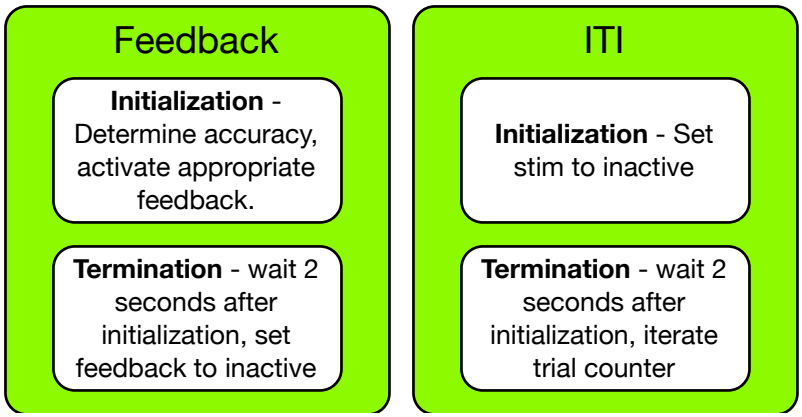


Figure 4. The Feedback and ITI states.

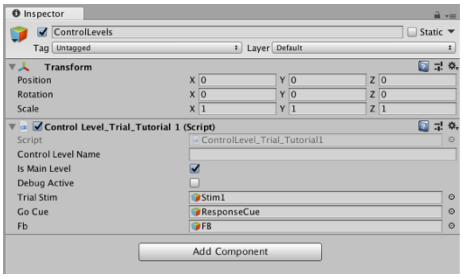
2.2.9 Specifying task termination and running the task

The final line of code we need to complete our task definition is:

```
        AddControlLevelTerminationSpecification(() => trialCount > 5);
```

This defines the termination criterion for the task as a whole – in this case, it will finish after 5 trials.

Now, we return to the Unity Editor. We are almost ready to run the task, the only thing that remains is to click on the *ControlLevels* object in the scene hierarchy, and click the "Is Main Level" checkbox for the *ControlLevel_Trial* script component.



This defines this Control Level as the primary Level in this task, which defines the control flow of the entire experiment. Each USE project must have one, and only one, main level, and they will not run at all if this is not the case.

At this point, we can finally click the play button and test our new task!

2.2.10 Troubleshooting and expanding the task

There is a completed Tutorial 1 found in the Tutorials_Completed package. Import this the same way you imported the USE_Core package (either by right-clicking one the Assets folder in the project tab and selecting Import Package / Custom Package, or by dragging and dropping directly from the file system.)

The folder you have just imported contains a Tutorial 1 folder, with a scene and a ControlLevel_Trial_Tutorial1_completed script that should be identical to the scene and script that you have created. If your project is not behaving as expected, try switching to this scene and seeing if you can find any differences between it and your scene, or between the completed script and yours.

There is also a ControlLevel_Trial_Tutorial1_extended script and scene. A number of changes have been made, which we do not review in detail, as they involve no USE-specific details, but it is recommended that you both read the code—changes have been highlighted with comments—and try the altered task, because it will be used in the next tutorial.

The changes include:

- There are now two objects on each trial.
- Feedback is probabilistic: one object is rewarded 85% of the time, the other 15%.
- The rewarded object is determined by means of a tag.
- Durations have been substantially reduced.
- A variable keeps track of the number of correct responses.
- Durations, number of trials, positions, etc, are now set by public variables, that can be edited in the editor or in by other scripts.

To run the altered task, simply switch to the extended scene and press play.

3 Tutorial 2 – Multiple Control Levels

3.1 Description

In this tutorial, we will expand on the task generated in Tutorial 1 by placing the trials into a block structure. Trials will use the ControlLevel_Trial_Tutorial1_extended task, with two objects, probabilistic reward, and changing object locations.

The important element of USE this tutorial introduces is the use of multiple hierarchical Control Levels. This is important enough that we introduce it without adding other new USE components, since using Control Levels in this way is the heart of USE's power and flexibility.

This tutorial, as with later ones, does not include screenshots or detailed descriptions of steps that repeat elements from Tutorial 1, unless they are introducing some new functionality.

3.2 Experiment Creation

3.2.1 Overview

We will copy the Tutorial 1 extended trial Control Level, and then create a new Control Level that handles block-specific information. At the end of each block of trials, participants will be given information concerning their performance, and the object with the higher reward value is randomly chosen. There are only two states to this Level: a Trial State, that will pass control to the Trial Control Level, and a Feedback State that will present the feedback screen.

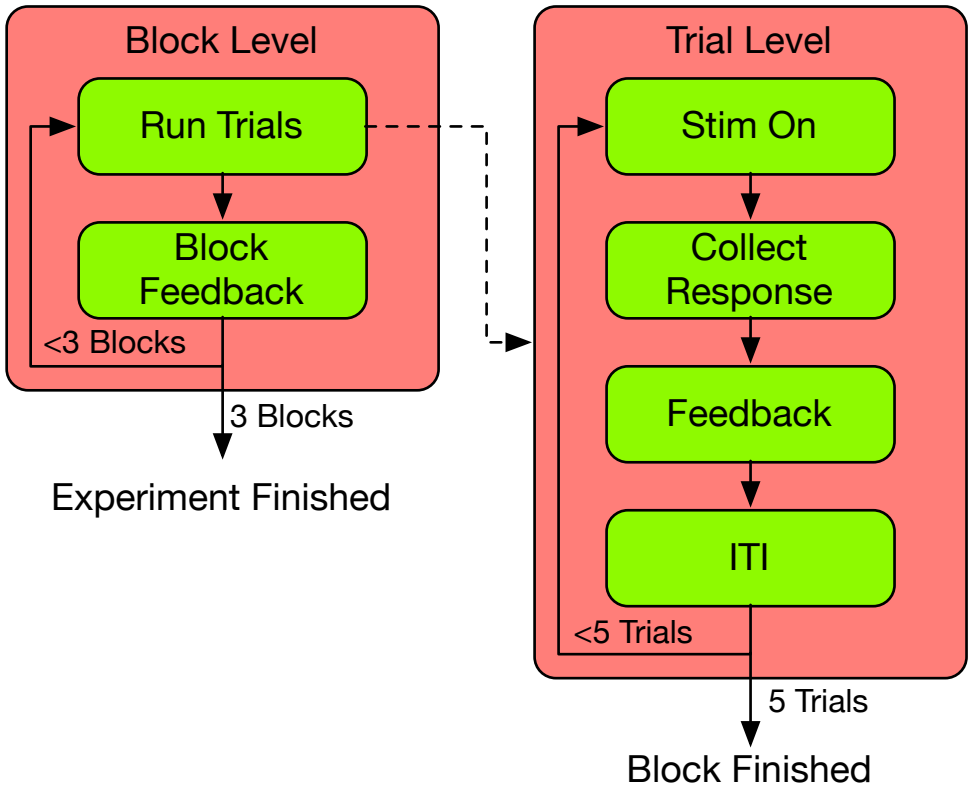


Figure 5. An overview of the Control Levels and States used in Tutorial 2. Note that the Block Level's Run Trials State passes control to the Trial Level, which actually manages the trials.

3.2.2 Copying the extended trial, adding new Control Level

Copy the contents of the ControlLevel_Trial_extended script (in the Tutorials_Complete folder) and replace the contents of your ControlLevel_Trial script. Make sure to change the name of the class to ControlLevel_Trial at the class definition line:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using USE_States;

public class ControlLevel_Trial_Tutorial1_extended : ControlLevel
{
```

Click on the ControlLevels object in the scene hierarchy again, and look at the Inspector. Note that the ControlLevel_Trial component is there, but four of its fields are empty. Populate them by dragging the appropriate objects from the scene hierarchy to these fields. Stim1 and Stim2 are the sphere and cube stims (the order is irrelevant), and the Go Cue and Fb are the same as in Tutorial 1.

Now create a new script called ControlLevel_Block. As in Tutorial 1, change this script by including the `USE_States` and `UnityEngine.UI` namespaces, changing the class' inheritance to the `ControlLevel` class instead of `MonoBehavior`, and delete the `Start` and `Update` methods. In the Unity Editor, drag the new Control Level script onto the ControlLevels object in the scene hierarchy. In the ControlLevels inspector, rename it BlockLevel, click *Is Main Level*, and un-click this for the TrialLevel component.

3.2.3 Defining the trial block

Open the new block level script. Declare public `stim1` and `stim2` GameObjects, `numBlocks`, `numTrials` and `currentBlock` ints, initialize the ControlLevel, create `runTrials` and `blockFb` States, and add them to the Control Level's active States:

```
public class ControlLevel_Block : ControlLevel
{
    public GameObject stim1;
    public GameObject stim2;


    public int numBlocks = 3;
    public int numTrials = 20;


    public int currentBlock = 1;

    public override void DefineControlLevel()
    {

        //initalize this Control Level
        InitializeControlLevel("CtrlLvl_Block");

        //define States within this Control Level
        State runTrials = new State("RunTrials");
        State blockFb = new State("BlockFB");

        AddActiveStates(new List<State> { trial, blockFb });
    }
}
```

Now, we want to make the `ControlLevel_Trial` object the child of the `runTrials` State. First, we need to access the object, using the `GetComponent` method. Since both the Block Level and the Trial Level are components of the same ControlLevels object in the hierarchy, they both share the same Transform, so we can refer to the TrialLevel using `transform.GetComponent`. Then, we use the `AddLevel` method to add this Level as a Child of the `runTrials` State:

```
ControlLevel_Trial trialLevel = transform.GetComponent<ControlLevel_Trial>();
runTrials.AddLevel(trialLevel);
```

`AddLevel` is a deceptively simple function, with far-reaching effects. When a child Control Level is added to a parent State using `AddLevel`, the components of the child's update cycle are run after those of the parent. Specifically, after each of the parent State's FixedUpdates, the FixedUpdate of the current State of the child Control Level will run, and the same is true of Update and LateUpdate methods. (It is possible to specify methods in a State to be run after its child, but this is not relevant in these tutorials.)

Thus, in this case, whatever State the `ControlLevel_Trial` is in will run its update methods on each frame, for as long as the `ControlLevel_Block` is in the `runTrials` State.

The `runTrials` State will initialize by setting the number of trials in a block, resetting the trial count and number correct, then determining which stimulus rewarded will be rewarded. It will terminate as soon as the Trial Control Level terminates, which is to say after all trials in the block have been run:

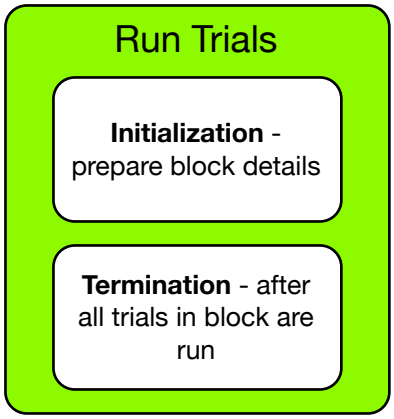


Figure 6. The Run Trials State. Note that it has no update method, but as the Trial Level is this Run Trials State's child, all the Trial Level's States will run while the Run Trials State is active.

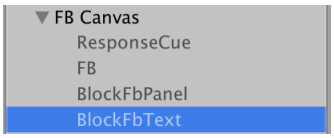
```
        runTrials.AddStateInitializationMethod(() =>
        {
            trialLevel.trialCount = numTrials;
            trialLevel.trialCount = 1;
            trialLevel.numCorrect = 0;


            if (Random.Range(0, 2) == 1)
            {
                stim1.tag = "Target";
                stim2.tag = "NotTarget";
            }else
            {
                stim1.tag = "NotTarget";
                stim2.tag = "Target";
            }
        });
        runTrials.SpecifyStateTermination(()=> runTrials.Terminated == true, blockFb);
```

3.2.4 Adding between-block feedback

We want to display a message to our participants between blocks. Start by declaring new `fbText` and fbPanel GameObjects, just after the stimulus GameObject declarations:

```
public class ControlLevel_Block : ControlLevel
{
    public GameObject stim1;
    public GameObject stim2;
    public GameObject fbText;
    public GameObject fbPanel;
```

Return to the Unity Editor, and create two new UI elements inside the FB Canvas object in the hierarchy: a text element called BlockFbText, and a Panel element called BlockFbPanel. The BlockFbPanel should be above the BlockFbText in the hierarchy, as position in the hierarchy determines display order of UI objects, and we want the text to display after the panel, so it shows up on top of it:



Drag these into their corresponding slots in the Inspector for the ControlLevel_Block component of the ControlLevels object, and set both to inactive (deselect the checkbox at the upper left of their Inspectors). Change the width of the BlockFbText to 800, and its height to 400. While we're making it look nicer, increase its text font size to 36 and center-justify it.

Now we need to populate the fields of the ControlLevel_Block object. Click on the ControlLevels object in the scene hierarchy, then drag the appropriate scene elements to the Stim1, Stim2, FbText and FbPanel fields of the Block script in the inspector (it does not make any difference whether the sphere is Stim1 and the cube Stim2, or vice versa).

Return to the script, and define the feedback State initialization, which makes the feedback canvas elements visible and prepares a custom message, which differs depending on overall accuracy (which it obtains from the trial Level) and whether this is the final block of the study. Nothing in this is new, from a USE perspective.

```
        blockFb.AddStateInitializationMethod(() =>
        {
            fbText.SetActive(true);
            fbPanel.SetActive(true);
            float acc = (float)trialLevel.numCorrect / (float)(trialLevel.trialCount - 1);

            string fbString = "You chose correctly on " + (acc * 100).ToString("F0") + "% of trials.\n
";
            if (acc >= 0.7)
            {
                fbString += "Nice Work!\n\nPress the space bar to ";
            }
            else
            {
                fbString += "You could probably get even higher! \n\nPress the space bar to ";
            }
            if (currentBlock < numBlocks)
            {
                fbString += "start the next block.";
            }
            else
            {
                fbString += "finish the experiment.";
            }

            fbText.GetComponent<Text>().text = fbString;
        });
```

There are three things we want to do at the end of this feedback State: make both the feedback canvas elements invisible, and iterate the block counter. To do this, we create a new method, outside the DefineControlLevel method.

```
    private void EndBlock()
    {
        fbText.SetActive(false);
        fbPanel.SetActive(false);
        currentBlock++;
    }
```

Then we call this new method as part of the termination specification. Remember this should be back inside the DefineControlLevel method:

```
blockFb.SpecifyStateTermination(() => InputBroker.GetKeyDown(KeyCode.Space), runTrials, ()=> EndBlock(
));
```

Any custom method defined elsewhere can be added to a termination, update, or initialization State in this manner, which is often more convenient or understandable than adding all the lines in the State specification.

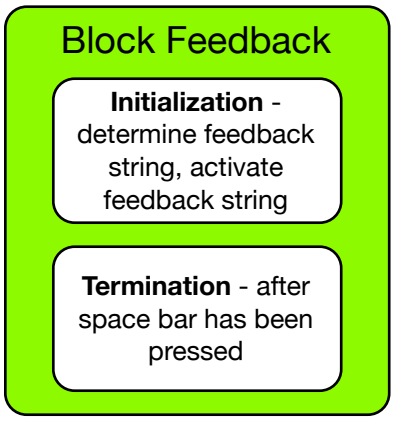


Figure 7. The Block Feedback State.

Finally, we add the Control Level termination specification:

```
        AddControlLevelTerminationSpecification(() => currentBlock > numBlocks);
```

And with this, we should be able to save, return to the Editor, and run the task.

Once again, there is a Tutorial_complete folder inside the Tutorials_complete folder you have already imported to the project. If the task does not work as expected, try comparing your version to this one.

4 Tutorial 3 – 3 Level Hierarchy

4.1 Description

From the participant's perspective, this task has no differences from that in Tutorial 2, save that there are now introductory and farewell screens. From the experimenter's point of view, the hierarchical organization of the task becomes both more complex and much more flexible in this tutorial.

4.2 Experiment Creation

4.2.1 Overview

This tutorial adds two more Control Levels to the experiment hierarchy. A introductory Slide Level that defines a series of instruction slides before the task, and a main Level that defines the overall experimental structure, passing control to the block Level (and thereby the trial Level).

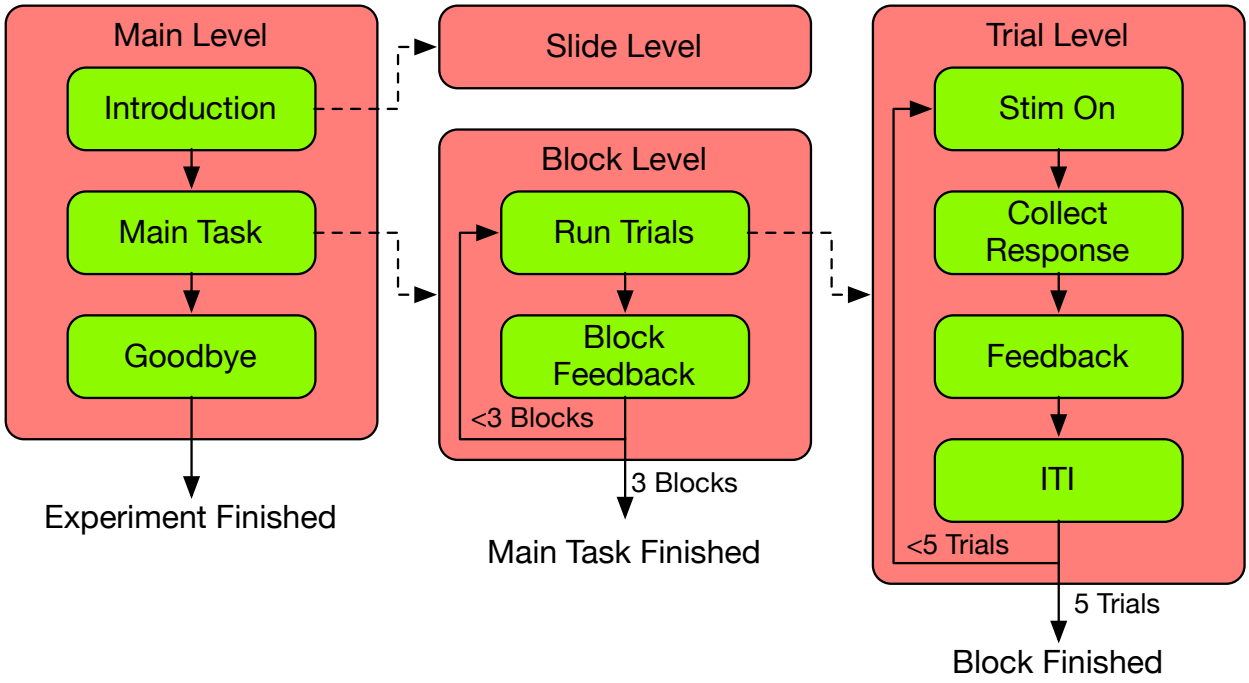


Figure 8. An outline of the structure of the Tutorial 3 experiment. The Main Level has two States with child Levels, one of which (the Block Level) has a State with its own child Level. The State details of the Slide Level are not shown as this is a pre-made level that automatically presents text slides.

4.2.2 Creating the new Control Level

In the Unity Editor, create a new script called ControlLevel_Main, and edit it to become a ControlLevel specification. Add the `UnityEngine.UI` and `USE_States` namespaces. Add two public GameObject variables for text display. Finally, give this Control Level three States called Intro, MainTask, and Goodbye:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using USE_States;

public class ControlLevel_Main_Tutorial3_complete : ControlLevel {

    public GameObject textObj, panelObj;

    public int numBlocks = 3, trialsPerBlock = 10;

    public override void DefineControlLevel(){
        State intro = new State("Intro");
        State mainTask = new State("MainTask");
        State goodbye = new State("Goodbye");
        AddActiveStates(new List<State> { intro, mainTask, goodbye });
    }
}
```

We are going to be defining the number of blocks and trials per block from this main level. In general, it is ideal to have all variables that control experimental parameters such as these to be defined at the main experimental level, and their values passed down to lower levels. This means that it is simple for experimental developers to find where these parameters are set, otherwise you will quickly end up with configuration variables being set in many different scripts, which is a nightmare for later editing (trust us!). Thus we will modify the Block Level slightly to have public variables without specified values, that are hidden from view in the Unity Editor (IMPORTANT: these changes are being made in the ControlLevel_Block script!):

```
public class ControlLevel_Block: ControlLevel
{
    public GameObject stim1;
    public GameObject stim2;
    public GameObject fbText;
    public GameObject fbPanel;

    [HideInInspector]
    public int numBlocks, numTrials, currentBlock = 1;

    public override void DefineControlLevel()
    {
```

Back in the Editor, drag the script onto the ControlLevels object in the scene hierarchy, and in the Inspector for the ControlLevels object and check its *Is Main Level* box (uncheck the same box for the Block Level). We will reuse the same canvas elements as in the previous tutorial. Drag the FbText and FbPanel canvas objects onto the script's Text Obj and Panel Obj properties. In the Control Level Name field, enter "Main Level".

4.2.3 Adding instructional slides

In the USE_Core/USE_ControlLevels folder in the Project tab, there is a script called ControlLevel_TextSlides. This is a pre-made control level suitable for displaying simple text slides for instructions. Drag it onto the ControlLevels object in the scene hierarchy, and populate its Text Obj and Panel Obj field with the FbText and FbPanel objects, just as you did for the Main Level.

Now return to the Main Level script and add the following:

```
        ControlLevel_TextSlides slideLevel = transform.GetComponent<ControlLevel_TextSlides>();

        slideLevel.slideText = new string[] {"Welcome to our study!\nThank you very much for participa
ting.",
            "In this task you will be shown two objects on each trial. You will have to choose one of
them by clicking on it with the mouse.",
            "Wait for the \"Go!\" signal before clicking.", "After clicking, you will get feedback. A
green square means your choice was rewarded. A red square means it was not.",
            "Try to learn which object gives the most reward.", "Ask the experimenter if you have any
questions, otherwise we will begin the experiment."};

        intro.AddChildLevel(slideLevel);
        intro.SpecifyTermination(() => slideLevel.Terminated, mainTask);
```

Most of this logic is identical to the way we added the Trial Level as a child of the Block Level's RunTrials State. The only additional thing to consider is that the TextSlides Control Level expects an array of strings to be supplied to its slideText field. Each string in the array will be displayed until the user presses the space button, at which point the next string will be shown, or, if there are no more strings, the Control Level will terminate.

4.2.4 Adding the Block and Goodbye Levels

We add the Block Level as a child of the MainTask State, and then add a simple goodbye message to display for two seconds after the blocks are completed:

```
        ControlLevel_Block_Tutorial3_complete blockLevel = transform.GetComponent<ControlLevel_Block_T
utorial3_complete>();
        blockLevel.numBlocks = numBlocks;
        blockLevel.numTrials = trialsPerBlock;
        mainTask.AddChildLevel(blockLevel);
        mainTask.SpecifyTermination(() => blockLevel.Terminated, goodbye);

        goodbye.AddInitializationMethod(() =>
        {
            textObj.SetActive(true);
            panelObj.SetActive(true);
            textObj.GetComponent<Text>().text = "Thank you very much for your time!";
        });
        goodbye.AddTimer(2f, null);
```

Note the use of the `null` in the `goodbye.AddTimer` method. This means that there will be no following state, and the experiment will terminate.

Once again, there is a complete version of the scene and scripts for you to compare your version to, if you encounter any issues.

5 Tutorial 4 – Data Management

5.1 Description

Nothing at all changes from the participant's perspective in this tutorial, and we do not add any States or Control Levels. However a critical aspect of the experimenter's trade is introduced: saving data. Specifically, we will create two data files, one where each line represents the important data from a given trial, and one where each line summarizes a given block.

5.2 Experiment Creation

5.2.1 Creating the DataControllers

Create a new empty object in the scene hierarchy, and name it DataControllers. Give it two new script components, named DataController_Block and DataController_Trial. Both of these scripts should be edited to include the `USE_Data` namespace, and to inherit from the `DataController` class. Furthermore, just as all Control Levels need to include an overridden DefineControlLevel method, all DataControllers need to include an overridden DefineDataController method:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using USE_Data;

public class DataController_Block : DataController

{

    public override void DefineDataController()
    {
    }
}
```

DataController is a powerful class that automates a great deal of experimental data handling. It outputs tab-delimited text files where each column represents a particular variable of interest (or output of a function), and each line contains the value of each of the variables at a particular moment in the experiment.

5.2.2 Defining Block Data

For the block data file, we want to keep track of a number of things. What block number are we currently in? Which trials were part of this block? What was the proportion of correct answers on these trials, and what was the proportion of rewarded answers (recall that participants are given imperfect feedback, and so may be unrewarded for correct answers, or rewarded for incorrect ones). Finally, how long did the block take?

To know which trials are part of each block, we will add public variables to the Block Level that track this information. First, we declare `firstTrial` and `lastTrial` variables (IMPORTANT: this is in the Block Level file):

```
    [HideInInspector]
    public int numBlocks, currentBlock, numTrials, firstTrial, lastTrial;
```

The `firstTrial` will be assigned its value in the runTrials.AddInitializationMethod, when the block begins:

```
        runTrials.AddInitializationMethod(() =>
        {
            trialLevel.numTrials = numTrials;
            trialLevel.trialInBlock = 1;
            trialLevel.numCorrect = 0;
            trialLevel.numReward = 0;
            firstTrial = trialLevel.trialInExperiment;
```

And the lastTrial will be assigned its value in the runTrials.SpecifyTermination, when the block ends:

```
        runTrials.SpecifyTermination(() => trialLevel.Terminated == true, blockFb, () => lastTrial = t
rialLevel.trialInExperiment);
```

Now we return the the DataController_Block file, and define the information we want to record. The block data controller will need access to variables stored in both the block and trial Control Levels. So we begin the `DefineDataController` method by providing references to those Levels (IMPORTANT: this is in the DataController_Block file):

```
        ControlLevel_Block blockLevel = GameObject.Find("ControlLevels").GetComponent<ControlLevel_Blo
ck>();
        ControlLevel_Trial trialLevel = GameObject.Find("ControlLevels").GetComponent<ControlLevel_Tri
al>();
```

Here we use `GameObject.Find` and `GetComponent` methods to reference these Levels. However we could also have made public fields of the `ControlLevel_Block` and `ControlLevel_Trial` types, and populated them by dragging the Control Level objects into their fields in the Unity Editor. Either method is equivalent (and there are other ways of accomplishing the same goal). By using this script-based method, however, we protect against references being lost in the Unity Editor when scripts or objects are copied, a problem you may have already encountered in previous tutorials. The downside is that we cannot be sure at compile time that the GameObject ControlLevels actually exists, or that it has a Component called ControlLevel_Block or ControlLevel_Trial. We leave it to you to determine which method you prefer.

To include data in a data file, we use the AddDatum method:

```
        AddDatum("Block", () => blockLevel.currentBlock);
        AddDatum("FirstTrial", () => blockLevel.firstTrial);
        AddDatum("LastTrial", () => blockLevel.lastTrial);
        AddDatum("Proportion Correct", () => trialLevel.numCorrect / trialLevel.numTrials);
        AddDatum("Proportion Reward", () => trialLevel.numReward / trialLevel.numTrials);
```

AddDatum has two arguments, first a string which will be used as the title of the data column, and then the actual variable or function to be tracked. As with adding methods to States, we use the lambda operator (`() =>`) in order to ensure that we will have access to the changing values of each of these variables. Note that the final two datum definitions use simple functions rather than single variables. Otherwise all these AddDatum commands are essentially the same, each defines a column of the data file.

Finally, we want to include some timing information. The USE DataController class has a simple way of keeping track of the timing information of each State in a Control Level, using a method called AddStateTimingData:

```
        AddStateTimingData(blockLevel, new string[] { "Duration"});
```

`AddStateTimingData` requires a Control Level as its first argument, which specifies which level's timing will be tracked. Here we also include an optional second argument, which specifies which timing information we wish to track for each state. This needs to be in the form of an array or list of strings, as there are multiple possible things we might include. Here, for

each State in the Block Level, we will only record its duration. If we do not include the optional argument, we would also record Start and End frame numbers for each state, as well as their Start and End times in two ways: relative to the start of the experiment, and relative to a user-specified starting time which can be updated during the experiment. However we have no particular interest in recording this information, and so we record only the Duration here. Note that since the Block Level has two States, we will record two durations for each block.

5.2.3 Defining Trial Data

There are more variables to record at the Trial Level, but doing so does not introduce any new USE-specific coding principles. Thus we present the entire contents of the DataController_Trial's DefineDataController method at once:

```
        blockLevel = GameObject.Find("ControlLevels").GetComponent<ControlLevel_Block_Tutorial4_comple
te>();
        trialLevel = GameObject.Find("ControlLevels").GetComponent<ControlLevel_Trial_Tutorial4_comple
te>();
        AddDatum("Block", () => blockLevel.currentBlock);
        AddDatum("TrialInBlock", () => trialLevel.trialInBlock);
        AddDatum("TrialInExperiment", () => trialLevel.trialInExperiment);
        AddDatum("Response", ()=> trialLevel.response);
        AddDatum("Reward", () => trialLevel.reward);
        AddDatum("Stim1_name", () => trialLevel.stim1.name);
        AddDatum("Stim1_targetStatus", () => trialLevel.stim1.tag == "Target" ? 1 : 0);
        AddDatum("Stim1_worldX", () => trialLevel.stim1.transform.position.x);
        AddDatum("Stim1_worldY", () => trialLevel.stim1.transform.position.y);
        AddDatum("Stim1_worldZ", () => trialLevel.stim1.transform.position.z);
        AddDatum("Stim1_screenX", () => Camera.main.WorldToScreenPoint(trialLevel.stim1.transform.posi
tion).x);
        AddDatum("Stim1_screenY", () => Camera.main.WorldToScreenPoint(trialLevel.stim1.transform.posi
tion).y);
        AddDatum("Stim2_name", () => trialLevel.stim2.name);
        AddDatum("Stim2_targetStatus", () => trialLevel.stim2.tag == "Target" ? 1 : 0);
        AddDatum("Stim2_worldX", () => trialLevel.stim2.transform.position.x);
        AddDatum("Stim2_worldY", () => trialLevel.stim2.transform.position.y);
        AddDatum("Stim2_worldZ", () => trialLevel.stim2.transform.position.z);
        AddDatum("Stim2_screenX", () => Camera.main.WorldToScreenPoint(trialLevel.stim2.transform.posi
tion).x);
        AddDatum("Stim2_screenY", () => Camera.main.WorldToScreenPoint(trialLevel.stim2.transform.posi
tion).y);
        AddStateTimingData(trialLevel, new string[] { "Duration", "StartFrame", "EndFrame" });
```

There are a few things worth noting here. First, the target status of each stimulus is determined using the conditional operator `?`, which is set to return 1 if the stimulus has the tag "Target" and 0 if it does not. Second, we store both the stimuli's position in the worldspace and their position on the screen. The latter uses Unity's `WorldToScreenPoint` method. Finally, note that we include three different types of timing information: each state's duration, as we did with the block data, and also its first and last frame.

5.2.4 Initializing Data Controllers

In keeping with the principle that experiment parameters should be specified by the Main Control Level, we will make some modifications to the ControlLevel_Main script that specify as much of the data control characteristics as possible. First, we add two public variables:

```
public class ControlLevel_Main_Tutorial4_complete : ControlLevel {

    public GameObject textObj;
    public GameObject panelObj;

    public int numBlocks = 3;
    public int trialsPerBlock = 10;

    public string dataPath = "ADD DATA PATH HERE";
    public bool storeData = true;
```

The `dataPath` string should be specified to point to a desired location on your hard drive. Also, by using a `storeData` Boolean, we can easily stop recording data during debugging.

Next, still in the ControlLevel_Main script, we specify important data controller parameters in the `DefineControlLevel` method:

```
        DataController_Block blockData = GameObject.Find("DataControllers").GetComponent<DataControlle
r_Block>();
        DataController_Trial trialData = GameObject.Find("DataControllers").GetComponent<DataControlle
r_Trial>();
        blockData.storeData = storeData;
        trialData.storeData = storeData;
        blockData.folderPath = dataPath;
        trialData.folderPath = dataPath;
        blockData.fileName = "BlockData.txt";
        trialData.fileName = "TrialData.txt";

        blockLevel.blockData = blockData;
        trialLevel.trialData = trialData;
```

This specifies whether data will be recorded at all (`storeData`), the folder it will be recorded in (`dataPath`), and the name of the data file (the `fileName` fields). Finally, it specifies which data controllers will be accessible by the Block and Trial Levels. This final step should throw an error, as we have not yet edited these levels to incorporate data controllers.

We will now fix this error. In the ControlLevel_Block script, add a public block data controller:

```
public class ControlLevel_Block: ControlLevel
{
    public GameObject stim1, stim2, fbText, fbPanel;

    [HideInInspector]
    public int numBlocks, currentBlock, numTrials, firstTrial, lastTrial;
    [HideInInspector]
    public DataController_Block blockData;
```

Do the same in the ControlLevel_Trial script:

```
    [HideInInspector]
    public DataController_Trial trialData;
```

The errors in the Main Level should now be taken care of. All that remains is to actually control the reading and writing of data in the Block and Trial Levels.

5.2.5 Adding AppendData and WriteData methods

Most activity of DataControllers is governed by two methods: `AppendData` and `WriteData`. `AppendData` goes through all the variables tracked by the DataController, and adds their current values to a buffer. `WriteData` appends this buffer to a file on the computer, and then empties the buffer. By default, if the buffer contains more than 100 lines, `WriteData` is automatically called.

In the current task, `AppendData` and `WriteData` are always called together, however this is not always desirable. In future tutorials, we will see examples of data that is updated multiple times

per second using the `AppendData` method, but which is written far less often. For example, we want to append the position of moving objects once per frame, but we will only write these data once per trial, to avoid flooding the system with too-frequent write commands.

In the ControlLevel_Block script, we simply add the two methods to the EndBlock method:

```
    private void EndBlock()
    {
        fbText.SetActive(false);
        fbPanel.SetActive(false);
        blockData.AppendData();
        blockData.WriteData();
        currentBlock++;
    }
```

This means that every time the EndBlock method is called (which occurs during the termination of the blockFb State), the current values of each of the `Datum` objects defined in the DataController_Block script will be appended to the data buffer, and this will be written to the block data file.

Similarly, we add the same two methods to the Trial Level's iti State termination method:

```
        iti.AddTimer(itiDur, stimOn, () => { trialInBlock++; trialInExperiment++; trialData.AppendData
(); trialData.WriteData(); });
```

Thus at the end of each trial, a new line of trial data will be added to the trial data file.

At this point, your experiment should run as in the previous tutorial, but keep track of data of interest. Check that the two data files are being added to the path you specified, and verify that their data is interpretable. As with previous tutorials, a complete scene and scripts are provided.

5.2.6 Timing data note

There is a wrinkle in State timing data that does not affect commands but is worth noting to understand the resulting data. The duration of a State is defined as the time from the onset of its first frame to the offset of its final frame. However this means that this duration cannot be calculated during this final frame, and thus is actually calculated at the start of the following frame. This is automatically taken care of by the DataController class: if you request it to track State durations, it delays appending and writing these data until the frame after the State ends, but all other data types are appended on the requested frame.

6 Tutorial 5 – Configuring experimental sessions with initialization screens and text files

6.1 Description

Once again, the participant's task does not change in this tutorial, and we do not add States or Control Levels. Here, we are concerned with aspects of an experiment that might change between sessions. For example the session number, subject identifier, or experimental condition might all change from one session to the next. It is not appropriate to make these changes in code, because we want this to remain stable throughout an experiment.

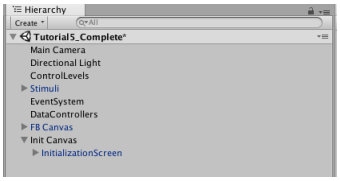
USE has three ways to configure experimental sessions. First, an initialization screen that appears at the beginning of an experiment, which displays fields that can be directly edited. In this tutorial, we will use this to input a subject identifier, and select two file paths: one to a data storage folder, and one to a configuration text file. The second way of configuring experimental sessions is using text files such as the one that our initialization screen points to. We will use one of these files to configure experimental variables such as the duration of different trial epochs, the number of trials in a block, and the number of blocks in a session. Finally, the third way of configuring a session is using a JSON file, which allows variables of any class, regardless of complexity, to be stored, provided an appropriate class is defined elsewhere in the experiment. These are beyond the scope of the present tutorials.

In general, the initialization screen is only used for fairly simple configurations (among other issues, at present it only allows for strings to be input), and to identify paths. The standard text files and JSON files are used for configuring many different variables, and allow for more complex possibilities.

6.2 Experiment creation

6.2.1 Preparing the initialization screen

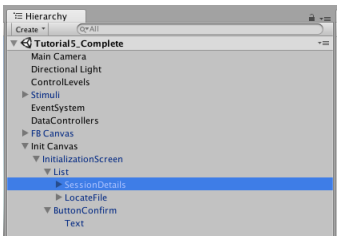
In the scene hierarchy tab, right-click and select UI/Canvas to create a new canvas. Call this Init Canvas. In the Project tab, navigate to _Scripts/USE_Modules/Initialization Screen. Drag the InitializationScreen prefab to the scene hierarchy and make it a child of the Init Canvas.



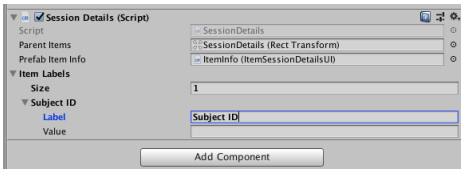
Now click on the ControlLevels object, and drag the InitializationScreen from the scene hierarchy to the Init Screen field of the ControlLevel_Main component. Do not drag the prefab from the Project tab to this screen – make sure it is the new object you have just created in the scene hierarchy. (Alternatively, click on the selection button to the right of this field and select the InitializationScreen through the ensuing dialogue.)

By adding an InitializationScreen object to a Control Level, a new State is added to the Level that runs prior to all others. It terminates when the user presses a confirmation button, and the user-defined States begin.

Initialization Screens can display two lists with user-definable elements – one of strings, one of file paths. We will add a string for the subject identifier. Expand the Initialization Screen object in the hierarchy, and expand the List object inside it. The List contains two objects: SessionDetails and LocateFile. Start by clicking on SessionDetails.

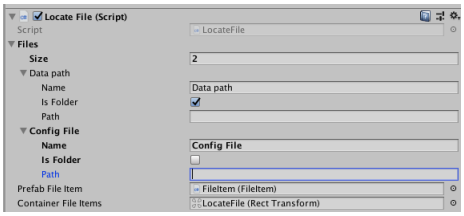


In the Inspector for the SessionDetails object, there is a script component with an Item Labels field. Set the size of this to 1, and an Element 0 will appear below this. Enter “Subject ID” into the Label field, and the Element 0 will change its name to the same string.

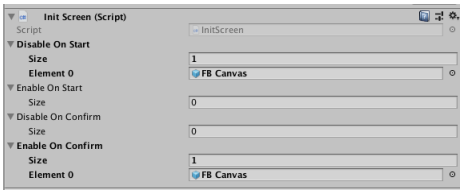


Now when the experiment starts, there will be a single field for a string to be entered, with the label “Subject ID”.

Entering file paths works similarly. Click on the LocateFile object in the hierarchy, and find the Locate File script component in the inspector. Change the Size field of the Files list to 2, and Elements 0 and 1 will appear below. Change the name of the first to “Data Path”, and select the Is Folder checkbox. Change the name of the second to “Config File”, and do not select the checkbox.



One final piece of setup is necessary for the initialization screen. Select it in the scene hierarchy, and in the Init Screen script component in the inspector, change the size of the Disable On Start and Enable on Confirm fields to 1. Add the FB Canvas to both lists.



Now when the experiment starts, the FB Canvas will be disabled and thus will not be able to block the Init Canvas. Once the user presses “Confirm” on the initialization screen, however, the FB Canvas will be re-enabled.

6.2.2 Preparing a configuration file

There are quite a few variables in the experiment whose values are hard-coded. We want these to be accessible to the experimenter without having to re-code the experiment, so we will store them in a text file. Open a text editor (e.g. TextEdit on a Mac, Notepad on Windows). Save the file as raw text to an appropriate location, with a name like "TutorialConfig.txt".

Lines in a config file that begin with two forward slashes ("//", with no quotation marks) will be ignored, so you can add comments to your files. Each variable takes one line, and consists of three elements, separated by tabs. First, the variable type (string, int, etc). Second, the name you wish to assign, and third, the value. Currently the config file reader will accept the four basic variable types in C# (string, bool, int, float), lists of all these types, arrays of all these types, and a number of other custom classes. It is easy to add more if needed. For this tutorial, we only need the basic types. We want to add all the variables whose values are currently hard-coded. The result should look something like this:

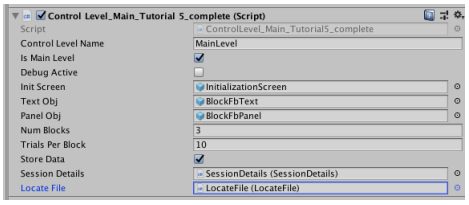
```
//Trial variables
float	stimOnDur 1
float	responseMaxDur 5
float	fbDur	0.5
float	itiDur	0.5
float	posRange	3
float	minDistance	1.5
float	rewardProb	0.85

//Block variables
int	numBlocks 3
int	numTrials 10

//General variables
Bool storeData true
```

6.2.3 Accessing configuration information in scripts

Add public SessionDetails and LocateFile objects to the Main Level:

```
    public SessionDetails sessionDetails;
    public LocateFile locateFile;
```

These need to be populated in Unity's inspector, either by dragging their corresponding objects from the scene or clicking the button to the right of the fields and using the selection dialogue.



We will now add an Initialization method to the first State in the Main Level, accessing the values from the Initialization Screen and the configuration file, and applying these to the appropriate variables:

```
        intro.AddInitializationMethod(() =>
        {
            string dataPath = locateFile.GetPath("Data Path");
            string configPath = locateFile.GetPath("Config File");
            ConfigReader.ReadConfig("exptConfig", configPath);

            bool storeData = ConfigReader.Get("exptConfig").Bool["storeData"];
            blockData.storeData = storeData;
            trialData.storeData = storeData;
            blockData.folderPath = dataPath;
            trialData.folderPath = dataPath;
            string participantID = sessionDetails.GetItemValue("Participant ID");
            blockData.fileName = participantID + "_BlockData.txt";
            trialData.fileName = participantID + "_TrialData.txt";
            blockData.CreateFile();
            trialData.CreateFile();



            blockLevel.numBlocks = ConfigReader.Get("exptConfig").Int["numBlocks"];
            blockLevel.numTrials = ConfigReader.Get("exptConfig").Int["numTrials"];

            trialLevel.stimOnDur = ConfigReader.Get("exptConfig").Float["stimOnDur"];
            trialLevel.responseMaxDur = ConfigReader.Get("exptConfig").Float["responseMaxDur"];
            trialLevel.fbDur = ConfigReader.Get("exptConfig").Float["fbDur"];
            trialLevel.itiDur = ConfigReader.Get("exptConfig").Float["itiDur"];
            trialLevel.posRange = ConfigReader.Get("exptConfig").Float["posRange"];
            trialLevel.minDistance = ConfigReader.Get("exptConfig").Float["minDistance"];
            trialLevel.rewardProb = ConfigReader.Get("exptConfig").Float["rewardProb"];
        });


        intro.AddChildLevel(slideLevel);
        intro.SpecifyTermination(() => slideLevel.Terminated, mainTask);
```

The path provided by any LocateFile object can be accessed using the `GetPath` method. Note that the string argument must be exactly the same as the Name field in the LocateFile object. These data path strings can then be used however they are needed, in our case to point to the the configuration file to be read, and to set the Data Controllers' data paths.

We then use the `ReadConfig` method to parse the configuration text file. This takes two arguments, one a string that will be used to refer to the collection of parsed variables from a file, and the second a string pointing to the file itself. One can thus have multiple configuration files if desired, so long as they are all referred to by a unique string. In order to access the parsed variables, use the `ConfigReader.Get` method. This takes an argument giving the name of the collection, and then requires the variable type to be specified, followed by a string referencing the variable name given in the text file. This string must be identical to that in the file, or it will not work. In the above example, we access a Boolean, two Int values, and several floats.

Now we can remove all the hard-coding from the scripts themselves. So instead of this line in the Trial Level script:

```
public float stimOnDur = 1f, responseMaxDur = 5f, fbDur = 0.5f, itiDur = 0.5f, posRange = 3f, minDista
nce = 1.5f, rewardProb = 0.85f;
```

we can now have:

```
public float stimOnDur, responseMaxDur, fbDur, itiDur, posRange, minDistance, rewardProb;
```

Similarly, in the Main Level we can remove the public numBlocks, trialsPerBlock and storeData variables completely.

At this point, the experiment should be ready to run. Note that after entering any values in the Initialization Screen, these will be preserved as default values on subsequent runs.

7 Tutorial 6 – Moving Stimuli and a Frame Data File

8 Tutorial 7 – Dual Display Setup

10 Tutorial 8 – Connecting to an Eyetracker

11 Tutorial 9 – Connecting to the USE SyncBox