Lab 1: Introduction to the development environment and transformations

CSE 3541
Autumn 2021
Roger Crawfis

PRELIMINARY NOTES

For the labs in this class, you will use the Unity game engine and C#. Before starting the assignment below, do the following:

- Make sure you have a computer to use for this class (Windows or Mac – no Linux). The CSE Computer Labs do not have Unity on them (licensing issues).
- Install the free version of Unity from here (Links to an external site.). Note, it is best to download the Unity Hub. I am currently using 3.0.0-beta as of 8-20-2021. (see https://learn.unity.com/tutorial/project-configuration-with-unity-hub-1).
- Install the latest version of Unity (2021.1 or 2021.2).
- Make sure Visual Studio is installed (or Visual Code – but computer science students should use a real tool).
- Install the WebGL platform.
- For those interested in game development, I would start working on the Junior Programming Pathway at learn.unity.com. Carl is pretty entertaining at 2x speed 😊.
- Watch a video or two about the new Input System, particularly with C# events: https://www.youtube.com/watch?v=YHC-6I_LSos, or https://www.youtube.com/watch?v=kGykP7VZCvg.
- Brush up on your C# skills (or quickly learn C# - very similar to Java). See my Programming in C# course videos: https://web.cse.ohio-state.edu/~crawfis.3/cse459_CSharp/index.html

---

ASSIGNMENT

The basic task is to write a program that a character using events from an input system that allows various devices. This relatively short lab is mainly intended to get everybody comfortable with the development environment and method of submission for labs. Labs after this one will be longer, less prescriptive, and will have more implementation requirements.

First, create a new empty Unity project.
Set up new folders as discussed in class (Scenes, Scripts, Materials, Input).

Set the default namespace for any scripts to your LastnameFirstname.Lab1 (e.g., CrawfisRoger.Lab1), change the color space to Linear (Refer to:

https://docs.unity3d.com/Manual/LinearRendering-LinearOrGammaWorkflow.html).
Next, save the scene by selecting File->Save Scene. Include your name as part the scene file name (ex: CrawfisMovingObject). You will use the built-in play, pause, and step buttons to view your animation. You can delete the old SampleScene.

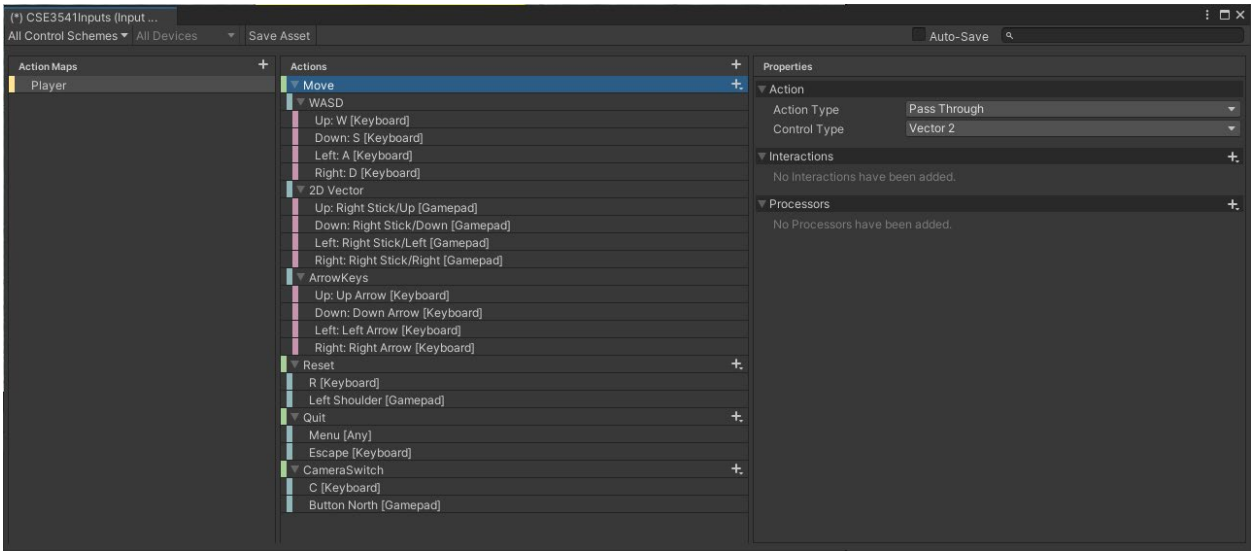
Unity Assets

Input Requirements

You will need to import the package Input System using the Package Manager (to: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Installation.html#installing-the-package). This will prompt you to replace the old input system, say yes and Unity will need to restart. Your input should consist of the following:

1) WASD (wasd) keys on the keyboard for movement (if you are not sure what this means, look it up).
2) Arrow keys on the keyboard for movement (same as WASD).
3) Gamepad left-stick for movement.
4) r key on keyboard and Left shoulder on gamepad to reset the position.
5) Esc on the keyboard and Menu button on the gamepad to quit.
6) c on the keyboard and North button (Y on Xbox One controller) to switch cameras.

Here is a completed screenshot:



You should use the 2D Vector as a composite type for WASD, arrow keys and Gamepad controls.

In the Inspector of the Input Action Asset, add a root namespace (LastNameFirstName.Input) and have it generate the C# class. You can look at this script. Most of it is JSON initialization (see: https://www.youtube.com/watch?v=YHC-6I_LSos).

Scene

Your scene should consist of an empty GameObject with a position of zero, no rotation and a scale of one. Set the name to *"Environment."* Children of this should be:

A plane going from (-25,0,0) to (25,0,100). Note a plane is 10 meters by 10 meters, so this implies a scale of (5,1,10). Name this *Floor*

Two cubes going along the z-axis of the plane on each x-axis edge. Name these *LeftEdge* and *RightEdge*.

Mark the root of the *Environment* node as **static** in the Inspector (and all of its children).

For the floor, you can add some simple textures. Import any packages you experiment with into a different Unity project and then only export (or copy over) 1 or 2 of these materials. They can get quite large. Here is a small useful one: https://assetstore.unity.com/packages/2d/textures-materials/gridbox-prototype-materials-129127. Without a texture or some context, it can be hard to tell your character is moving. Other choices:

https://assetstore.unity.com/packages/2d/textures-materials/stone/tileable-cobblestone-texture-21235

https://assetstore.unity.com/packages/2d/textures-materials/tiles/hand-painted-seamless-tiles-texture-vol-5-162143

Your submission should be less than 3MB!!!

Character

Now create a hierarchical character. Below are some examples. You can use an Empty GameObject as the base or one of the default Unity primitives. I would suggest an empty GameObject and will explain why when we reach collisions, but this gives it a local coordinate frame. Make sure the children are all above the plane. You can use the empty game object (give it a name, like Fred or Sally) to give the character an initial position. Some inspiration:

https://www.youtube.com/watch?v=PC7lj3_73nE

https://www.youtube.com/watch?v=8wm9ti-gzLM

https://www.youtube.com/watch?v=3VQl0scK5HM

https://www.artstation.com/artwork/ZWeZN (don't get this complicated, but you could make an entire voxel character).

A few that took me less than 3 minutes to make:

They are comprised of 2, 8, 8 and 12 primitives respectively. The third one is just some transformations of the second one. Later I will point you to some resources on how to make simple characters in Blender (or Maya). A topic not covered in this class.

In your Materials folder, create several basic materials. Note, for this lab, do not use pre-built models or textures for the character. Check out a color palette to help with nice looking material colors (hint avoid fully saturated colors (255,0,0), etc.). https://coolors.co/, https://blog.templatetoaster.com/color-palette-generators/.

Unity Scripts

I will make this exercise as explicit as I can. My sample implementation has 8 classes (5 inherited from MonoBehaviour and 3 other classes), with a few more classes implementing the extra credit. All are less than 30 lines of code, with only 1-11 lines of executable code! I will give a skeleton for most of these.

Quit Handler

Try to always add this as one of the very first things you do. Otherwise, you have a build that you cannot quit out of. Follow the tutorial here (https://www.youtube.com/watch?v=YHC-6I_LSos) for how to use C# events and subscribe to them. There are some more subtle issues I will cover in class if you are interested in a more professional approach (e.g., Moving vs MoveRequested, and QuitRequest vs Quitting events). This quit handler (above) will work for both the Editor as well as a finished built game when the quit event is fired. It simply calls *Application.Quit()* when outside a build or sets *isPlaying* to false if running in the Unity editor. Of course, it will not work yet in your scene, as

```
using UnityEngine;
using UnityEngine.InputSystem;

namespace CrawfisSoftware
{
    public class QuitHandler
    {
        public QuitHandler(InputAction quitAction)
        {
            quitAction.performed += QuitAction_performed;
            quitAction.Enable();
        }

        private void QuitAction_performed(InputAction.CallbackContext obj)
        {
#if UNITY_EDITOR
            UnityEditor.EditorApplication.isPlaying = false;
#endif
            Application.Quit();
        }
    }
}
```

we have not created an instance of this class. For this, we use a new class, InputManager (below).

Movement

You should write a script (MonoBehavior) to handle the move events. To the right is my skeleton. It takes a GameObject that it will move (indicated by the [SerializeField] *attribute*, as well as a movement speed (in m/s). These are assigned in the Unity Editor for each Scene / usage. To reduce coupling, an Initialization method

```
using UnityEngine;
using UnityEngine.InputSystem;

namespace CrawfisSoftware
{
  public class MovementControl : MonoBehaviour
  {
    [SerializeField] private GameObject playerToMove;
    [SerializeField] private float speed = 5f;
    private InputAction moveAction;

    public void Initialize(InputAction moveAction)
    {
…
    }
    private void FixedUpdate()
    {
…
    }
  }
}
```

```
using CrawfisSoftware;
using CrawfisSoftware.Input;
using UnityEngine;

namespace CrawfisSoftware
{
  public class InputManager : MonoBehaviour
  {
    [SerializeField] private MovementControl movementController;
    private CSE3541Inputs inputScheme;

    private void Awake()
    {
        inputScheme = new CSE3541Inputs();
        movementController.Initialize(inputScheme.Player.Move);
    }
    private void OnEnable()
    {
        var _ = new QuitHandler(inputScheme.Player.Quit);
    }
  }
}
```

is added which is passed in the InputAction associated with movement. Make sure you enable the InputAction and save a reference to it for the FixedUpdate method. Again, we need to create an instance of this class in order for our movement to work. This will be handled next.

Input Manager

Our InputManager class will handle all of the initialization of the input and associated handlers. It is what Unity and others would call the PlayerInput class. Below is the basic structure. We will add to this as the lab is worked out. Note that the class CSE3541Inputs is the name I called my InputActionMap asset, so this is the name it choose automatically we it generated the class. This class is now automatically recreated every time we make a change to the InputMap. Since this is derived from MonoBehaviour, we can add it to a GameObject in our scene to create it. Unity will then call its Awake and OnEnable methods when we start our game.

Cameras

For this project, we cannot see the characters face with an over the shoulder camera (or no rotations on the character or camera). We will add 4 cameras:

1) Typical 3rd person Rear-facing camera
2) Front-facing camera (rotate about y-axis 180 degrees).
3) Side camera
4) Top-down camera

For the 4th camera, make it an orthographic camera (if you need help: https://www.youtube.com/watch?v=9CjQKVtshno). Zoom in on your character some (except maybe for the top-down camera). If you play your "game" now, the character will move outside the camera. We will fix this with a script that moves the camera to follow the player. To do this, we will use a very simple script attached to **each** camera as shown on the left. Set the position of the gameObject (its transform) to the position of the target plus the offset. There is one line of code for you to add. You will need to set the offset vector in the Unity Editor for each camera.

```
using UnityEngine;

namespace CrawfisSoftware
{
    class FollowWithOffset : MonoBehaviour
    {
        [SerializeField] private Transform target;
        [SerializeField] private Vector3 offset;

        private void Update()
        {
…
        }
    }
}
```

```
using UnityEngine;

namespace CrawfisSoftware
{
  public class CameraSwitcher : MonoBehaviour
  {
    [SerializeField] private Camera[] cameras;
    [SerializeField] private Camera defaultCamera;
    private int index = 0;

    void Start()
    {
        index = 0;
        // Loop through each camera and disable it.
        // Enable the default camera
        // (optional) make sure next camera is
        // not the default (if more than one)
    }

    public void NextCamera()
    {
        // Enable the next camera
        // then disable the current camera
    }
  }
}
```

We also need a small piece of code (aren't they all nice and small!) to switch cameras. For this lab, we will cycle through an array or list of cameras. This will show you how easy it is to add arrays as [SerializeField]'s that can be set in the Unity editor. We will keep an index of the current camera and then activate the next camera each time the event is fired. For this to work, you need to enable the next camera **before** disabling the active camera. Here is a skeleton to get you started. Once you have this, you can initialize it in the Input Manager

Reset / Respawn Character

Write a script (MonoBehaviour) to store the initial position of the character on Awake or Start. I called mine PlayerRespawner, but Respawner would have been a better name. Add a method called Respawn, that will set the position back to the initial position. Also save off the rotation (a Quaternion) and reset it as well (transform rotation). Now write a ResetRequestHandler or just ResetHandler that is much like the QuitHandler in that it subscribes to the performed event of an InputAction passed in an Initialize

```
using CrawfisSoftware;
using CrawfisSoftware.Input;
using UnityEngine;

namespace CrawfisSoftware
{
  public class InputManager : MonoBehaviour
  {
    [SerializeField] private MovementControl movementController;
    [SerializeField] private CameraSwitcher cameraSwitcher;
    [SerializeField] private PlayerRespawner playerRespawner;
    private CSE3541Inputs inputScheme;
    private ResetHandler resetHandler;

    private void Awake()
    {
        inputScheme = new CSE3541Inputs();
        movementController.Initialize(inputScheme.Player.Move);
        resetHandler = new ResetHandler(inputScheme.Player.Reset, playerRespawner);
    }
    private void OnEnable()
    {
        var _ = new QuitHandler(inputScheme.Player.Quit);
        var nextCameraHandler = new NextCameraHandler(inputScheme.Player.CameraSwitch, this.cameraSwitcher);
    }
  }
}
```

method. In this case though, also pass in a PlayerRespawner (or Respawner) instance and store that. When the event fires, simply call Respawn. See the final InputManager on how this is initialized.

Input Manager (final)

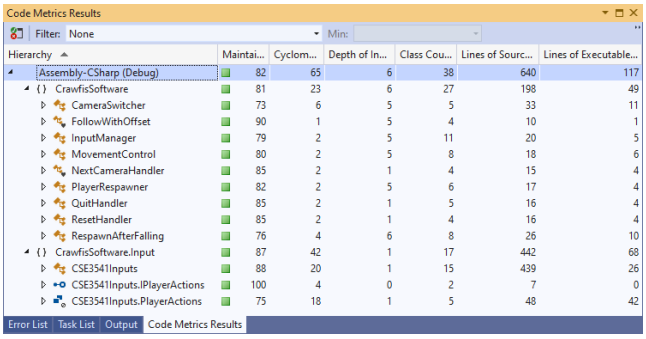
Now that we have a camera switcher, and a PlayerRespawner in addition to our QuitHandler and MovementController, we can create and / or initialize all of our input functionality. Note, this could be split across many classes. Here I have placed all of the “glue” needed to get our simple control scheme working in one class, so high cohesions, but somewhat higher coupling.

Documentation / Lab Report

Write up your experience in implementing this lab, include a few screen shots of your final product, and answer the following questions. Include all your source code in this report (except the autogenerated Input class). Submit this as a PDF file.

1) How hard or easy did you find this lab?
2) Have you ever worked in Unity before?
3) Have you programmed in C# much?
4) Did you do any of the extra credit? If so, explain in detail and include images.
5) What happens if you wire-up your Movement Controller to a part or child of your character?
6) What scripts could you reuse for other projects?
7) Which scripts could you not reuse?

8) What are the bugs you have found in this implementation / architecture? Hint: there are some. Any ideas on how to fix these?
9) Run Analyze->Code Metrics in Visual Studio and report the results (summarize and show a readable table). See example.
10) What are the scale values of children objects for you character? How does this work?
11) What happens when you move past the end of the plane (before and after the extra credit)?



Lab Submission

In addition to your report, submit a Unity package and a grade assessment.

Within Unity, right-click the scene file and choose Export Package … This ensures that only the files used in the scene are exported (as well as all scripts as it cannot tell sometimes). Save the package as YourlastnameLab1. This should result in a file called YourLastNameLab1.unitypackage being created in the folder you specified (hint, use Documents or some other space). This is the file you should turn in for grading.
Under the Lab1 assignment on Carmen, submit both the PDF file and the unitypackage file using File Upload option (not the Buckeye Box option).

You can test to determine if you have made the package correctly by creating a new Unity project, open the package file from a file browser, import all of the files in it, open the scene within Unity, and click play.

GRADING

Lab 1 will be graded out of 30 points, based on the following criteria:

- Is the lab submitted in the correct format and following our file naming conventions? (5 points)
- Was a thorough report submitted in addition to the lab implementation? (7 point)
- Is the Scene set-up properly with good names? (2 points)
- Is the motion control correct (orthogonal translation using the WASD keys) and can the character's position be reset (pressing R)? (5 point)
- Do the arrow keys and gamepad function the same as the keyboard? (2 points)
- Does the camera switching work as expected? (3 points)
- Is the student's name on the report and used in the namespaces? (1 point)
- How clean was the code? Were good detailed variable names used, etc.? (5 points)

Extra Credit

For extra credit (max 5 points) as well as greater complexity / challenge consider the following (each worth 5 points). Ambitious students should do all three!

A. Gravity, collisions and a better respawn.
    1) Add a Box Collider to the root of your character as well as a RigidBody. You may want to freeze the rotations. (1 point)
    2) Add a script that checks if a GameObject's y-position falls below a threshold (falls off of the plane). If so, it calls the Respawn (1 point)
    3) Rather than call Respawn right away, start a Coroutine that waits for a few seconds and then respawns. (2 points)
    4) Have the character respawn a couple of meters above the plane (use a SerializeField for this distance) (1 point)
B. Multiple characters and character switching
    1) Create 3 additional characters. (1 point)
    2) Write a script to switch between characters. (1 point)
    3) Add a new Input binding and event handler for switching characters (Left-Ctrl for previous player and Right-Control for next player, Dpad left/right in gamepad) (1 point)
    4) (option 1) Hide all other characters and move the next character to the current characters position. (2 points)
    5) (option 2) Have all characters visible and switch which one you are moving (and hence the camera) (2 points)
C. Local multiplayer
    1) Watch the Input Systems on local multiplayer
    2) Look at the included sample (within PackageManager->Input System) for local multiplayer.
    3) Create 2 additional characters. (1 point)
    4) Have WASD control one character, arrow keys control a second character and the gamepad a third character. (4 points)

Known Bugs

To be hidden from students at first.