# **SCRIPT DESIGN - Ah My 2 Legs**

**By:** Daniel Sohler

Contents

[SCRIPT DESIGN – Ah My 2 Legs 1](#_Toc69075429)

[Document overview 4](#_Toc69075430)

[Project Script Requirements 4](#_Toc69075431)

[Description 4](#_Toc69075432)

[Feature & Mechanics List 4](#_Toc69075433)

[Key Scripts to Design 4](#_Toc69075434)

[Script Design 5](#_Toc69075435)

[Script 1 – Player Controller 5](#_Toc69075436)

[Required Functionality & Outcomes 5](#_Toc69075437)

[Pseduocode 5](#_Toc69075438)

[Flowchart 5](#_Toc69075439)

[Script 1 Plan feedback 5](#_Toc69075440)

[Pseudocode feedback notes 5](#_Toc69075441)

[Flowchart feedback notes 5](#_Toc69075442)

[Script 1 Revised Plans 5](#_Toc69075443)

[Final pseudocode 5](#_Toc69075444)

[Final flowchart 6](#_Toc69075445)

[Script 2 – Energy Meter 6](#_Toc69075446)

[Required Functionality & Outcomes 6](#_Toc69075447)

[Pseduocode 6](#_Toc69075448)

[Flowchart 6](#_Toc69075449)

[Script 2 Plan feedback 6](#_Toc69075450)

[Pseudocode feedback notes 6](#_Toc69075451)

[Flowchart feedback notes 6](#_Toc69075452)

[Script 2 Revised Plans 6](#_Toc69075453)

[Final pseudocode 6](#_Toc69075454)

[Final flowchart 7](#_Toc69075455)

[Script Implementation & Iteration 7](#_Toc69075456)

[Script 1 – Player Controller 7](#_Toc69075457)

[Script(s) generated 7](#_Toc69075458)

[Functionality review 7](#_Toc69075459)

[Implementation feedback notes 7](#_Toc69075460)

[Response to feedback notes 7](#_Toc69075461)

[Script 2 – Energy Meter 8](#_Toc69075462)

[Script(s) generated 8](#_Toc69075463)

[Functionality review 8](#_Toc69075464)

[Implementation feedback notes 8](#_Toc69075465)

[Response to feedback notes 8](#_Toc69075466)

Document overview

The purpose of this document is to show the planning, design, implementation of at least two scripts in a single project. The document is broken up into phases to go through in order.

Project Script Requirements

### Description

Ah my 2 Legs is a 2D physics-based platformer, where the player is a mutilated prisoner, using a slingshot to escape the prison. Each level they collect keys, while avoid dangerous obstacles, and use the key to open the exit door. The player must also manage an energy meter, which limits how much they can slingshot themselves.

### Feature & Mechanics List

* **Slingshot Controls** – Main player controller, slings player around level.
* **Energy Meter** – Consumed when player slings themselves, manageable resource.
* **Key Inventory** – Holds keys used to unlock doors in levels.
* **Hazardous floors/walls** – Prevents player from easily manoeuvring terrain.

### Key Scripts to Design

* **Script 1** - Player Controller (Line trajectory script, Camera follow script, Timer script
* **Script 2** - Energy Meter – GUI script

## Script Design

### Script 1 - Player Controller

#### Required Functionality & Outcomes

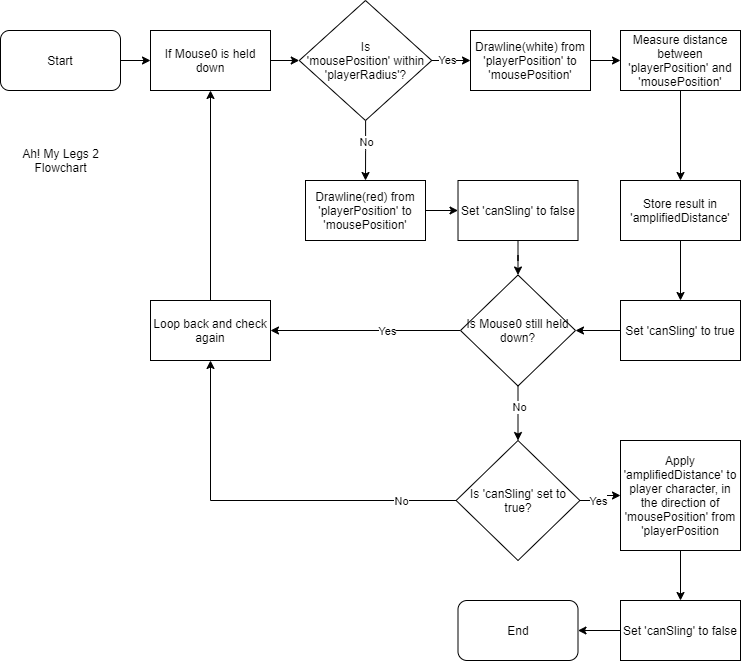
In this script, the controller needs to be able to track the mouse cursors position, and the player’s position. Using these two transforms, they will be used to calculate launch direction and distance. It will also prevent the player from launching if certain parameters aren’t met (right distance, available meter usage).

* Tracks when mouse is initially held down, constantly held down, and let go
* Tracks where on the screen the mouse is, and if its within a ‘safe’ radius of the player
* Draws a line from the player to the cursor to represent launch trajectory
* Prevents a launch if parameters aren’t met

#### Pseudocode

|  |
| --- |
| - Start - If GetMouse0Down is true, then  - If mousePosition is within playerRadius, then  - Drawline(white) from playerPosition to mousePosition, updating every frame  - Calculate distance between playerPosition and mousePosition into store into amplifiedDistance & update value every frame that GetMouse0Down is true.  - Set canSling to true   - else if mousePosition isn’t within playerRadius, then  - Drawline(Red) from playerPosition to mousePosition, updating every frame  - Set canSling to false - If GetMouse0Down was true, and is false and canSling is true   - If canSling is true, then  - Set slingshotFired to true  - Apply amplifiedDistance to player character, in the direction of mousePosition from playerPosition  - Set canSlign to false  - Set slingshotFired to false |

#### Flowchart



### Script 1 Plan feedback

#### Pseudocode feedback notes

* Bools don’t account for ‘was and is’, a separate Boolean would be needed.
* Add a check to see if the player properly releases the button.
* Reads too much like actual code than pseudocode.

#### Flowchart feedback notes

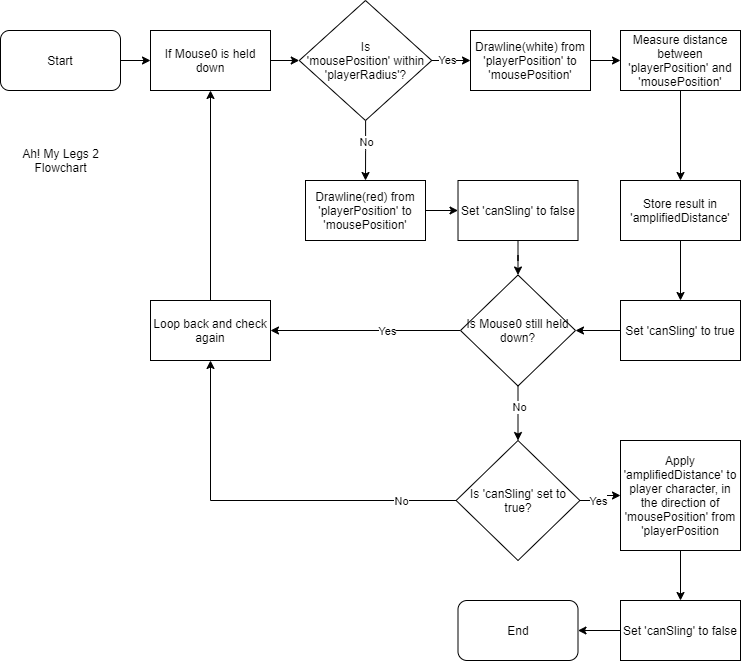
* N/A
* N/A
* N/A

### Script 1 Revised Plans

#### Final pseudocode

|  |
| --- |
| - Start - If GetMouse0Down is true, then  - Set mouseHeld to true,  - If mousePosition is within playerRadius, then  - Drawline(white) from playerPosition to mousePosition, updating every frame  - Calculate distance between playerPosition and mousePosition into store into amplifiedDistance & update value every frame that GetMouse0Down is true.  - Set canSling to true   - else if mousePosition isn’t within playerRadius, then  - Drawline(Red) from playerPosition to mousePosition, updating every frame  - Set canSling to false - If GetMouse0Down is false, mouseHeld is true, and canSling is true   - If canSling is true, then  - Set slingshotFired to true  - Apply amplifiedDistance to player character, in the direction of mousePosition from playerPosition  - Set canSlign to false  - Set slingshotFired to false  - Set mouseHeld to false. |

#### Final flowchart



### Script 2 - Energy Meter

#### Required Functionality & Outcomes

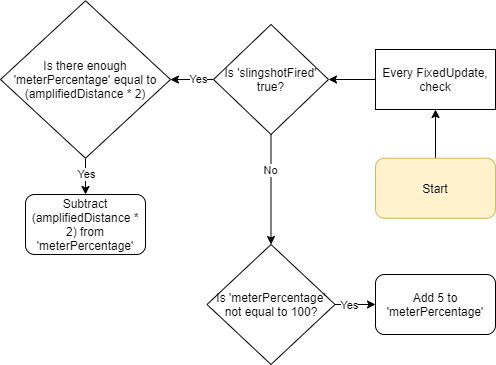
This meter located at the bottom-middle of the screen. This meter will be full at the start of the game. When the player launches themselves with the slingshot, the meter will have some percentage consumed.  
The meter will constantly refill itself over time at a slow rate.  
The meter also won’t be consumed if certain prerequisites aren’t met (available energy, canSling == true ect.)

* Constantly refils if not at a certain value (100 ect.)
* Prevents player controller from launching if lacking meter percentage.
* Consumes meter percentage when launched (correlating with launch velocity)

#### Pseudocode

|  |
| --- |
| - start - on FixedUpdate - If ‘slingshotFired’ is true, then  - If there is enough meterPercentage equal to (amplifiedDistance \* 2)?, then  - Subtract (amplifiedDistance \* 2) from meterPercentage  - if false, then  - If meterPercentage isn’t equal to 100, then  - Add 5 too meterPercentage every second |

#### Flowchart



### Script 2 Plan feedback

#### Pseudocode feedback notes

* There are two decision points that lack a ‘no’ scenario.
* N/A
* N/A

#### Flowchart feedback notes

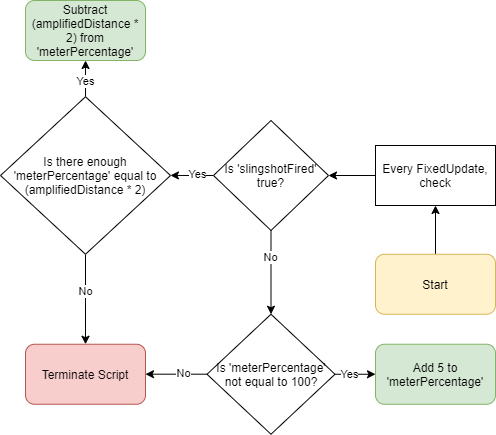
* Same as above
* N/A

### Script 2 Revised Plans

#### Final pseudocode

|  |
| --- |
| - start - on FixedUpdate (once every second instead of framerate) - If ‘slingshotFired’ is true, then  - If there is enough meterPercentage equal to (amplifiedDistance \* 2)?, then  - Subtract (amplifiedDistance \* 2) from meterPercentage  - If there isn’t enough meterPercentage equal to (amplifiedDistance \* 2), then  - Terminate  - if false, then  - If meterPercentage isn’t equal to 100, then  - Add 5 too meterPercentage every second  - if meterPercentage is equal to 100, then  - Terminate |

#### Final flowchart



## 

## Script Implementation & Iteration

### Script 1 – Player Controller

#### Script(s) generated

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  public class PlayerControllerScript : MonoBehaviour  {  // Variables needed  public Camera mainCam; // Uses main camera to get world position of mouse clicks later on  public Rigidbody playerCharacter; // reference to player rigidbody  public Text inputTrackingText; // refernce for where the 'inputTracker' int applies to on screen  public float power = 10f; // base number slings are calculated too  public bool canSling; // checks if its valid to sling  public LineTrajectoryScript lineTraj; // Calls other script here for line rendering  public float addedDirectionalForce; // Stores overall force of push  float addedDirectionalForceX; // Stores X value of mouseEndPoint and playerposition  float addedDirectionalForceY; // Stores Y value of mouseEndPoint and playerposition  public float minSlingDistance; // min required distance to draw line + allow the playr to launch themselves  public float maxSlingDistance; // max required distance to draw line + allow the playr to launch themselves  public bool requiredPower = true; // Connects to the 'EnergyMeterScript' to disable firing when not enough energy is present  public int inputTracker = 0; // Used for UI tracking of player inputs.  Vector3 currentMousePosition; // Actively tracked mouse position  Vector3 playerPosition; // tracks player position  Vector3 mouseEndPoint; //tracks where mouse is no longer held down  Vector3 launchDirection; // Impulse added to player character  public static PlayerControllerScript instance;  private void Awake()  {  instance = this;  }  private void Start()  {  // calls line trajectory script  lineTraj = GetComponent<LineTrajectoryScript>();  }  public void Update()  {  // Tracks current mouse position on every frame  currentMousePosition = mainCam.ScreenToWorldPoint(Input.mousePosition + new Vector3(0, 0, 12));  // Tracks players position, used for determining sling direction + line draw  playerPosition = playerCharacter.gameObject.transform.position;  // Min/Max launch distance code  // Minimum distance player can launch themselves too  if (Vector3.Distance(currentMousePosition, playerPosition) < minSlingDistance)  {  // If cursor is within minimum sling distance (i.e inside the player), then disable canSling  canSling = false;  }  else if (Vector3.Distance(currentMousePosition, playerPosition) > maxSlingDistance)  {  // If cursor is beyond the maximum sling distance, then disable canSling  canSling = false;  }  else  {  // If its within both the maxSlingDistance and the minSlingDistance, then set canSling to true  canSling = true;  }  // If its true, make the line blue  // If else, make the line red  if (canSling == true)  {  lineTraj.wantedLineRenderer.startColor = new Color(255, 255, 255);  lineTraj.wantedLineRenderer.endColor = new Color(0, 122, 255);  }  else  {  lineTraj.wantedLineRenderer.startColor = new Color(255, 255, 255);  lineTraj.wantedLineRenderer.endColor = new Color(255, 0, 0);  }  // While mouse is held down, render line between player position and currentMousePositions  if (Input.GetMouseButton(0))  {  lineTraj.RenderLine(playerPosition, currentMousePosition);  }  // Uses the players current position, and calculates the direction of the impulse  // by the position the mouse was before it was let go.  if (Input.GetMouseButtonUp(0))  {  // takes the end point the mouse cursor was at, used for calculating angle of impulse and distance  mouseEndPoint = mainCam.ScreenToWorldPoint(Input.mousePosition + new Vector3(0, 0, 12));  // minuses the mouseEndPoint by the playerPosition, then normalizes to return a smooth number that isn't a bloated vector (x and y seperate)  launchDirection = (mouseEndPoint - playerPosition).normalized;  addedDirectionalForceX = mouseEndPoint.x - playerPosition.x;  addedDirectionalForceY = mouseEndPoint.y - playerPosition.y;  // Removes negatives from force in X axis  if (addedDirectionalForceX <= 0)  {  addedDirectionalForceX = addedDirectionalForceX \* -1;  }  // Removes negatives from force in Y axis  if (addedDirectionalForceY <= 0)  {  addedDirectionalForceY = addedDirectionalForceY \* -1;  }  addedDirectionalForce = (addedDirectionalForceX + addedDirectionalForceY) \* power;  if (canSling == true && requiredPower == true)  {  playerCharacter.AddForce(launchDirection \* addedDirectionalForce, ForceMode.Impulse);  addedDirectionalForce = addedDirectionalForce \* 2;  EnergyMeterScript.instance.DrainPower(addedDirectionalForce);  inputTracker++;  TimerScript.instance.StartTimer();  }  // will end line even if canSling is set to false  lineTraj.EndLine();  // updates inputTracker UI element with amount of 'valid' inputs  inputTrackingText.text = ""+inputTracker;  }  }  } |

#### Functionality review

The script works as initially intended, while being notably different from the initial pseudocode. A main shortcoming, I wasn’t initially aware of was specificity that was needed for calculating the ‘addedDirectionalForce’. Initially I added the X/y’s without breaking up the values, which lead to unexpected inputs on screen. This was solved by separating the x/y of the player controller and mouse position during calculating ‘addedDirectionalForce’.

#### Implementation feedback notes

* Was easily digestible without knowing its prior functions.
* Use of RGB values instead of flat colours was unknown to some of them.
* Controller seemed very consistent for player inputs.
* Try adding an input counter, and a timer for tracking player progress and incentivising a competitive attitude to the game.

#### Response to feedback notes

* While the script is readable, I need to cut down on having excessive quoting throughout the script. It may bloat its overall file size, while being somewhat self-evident.
* Using the specific set of colours the .colours variable has are limiting, and it’d be good practice to get comfortable with the RGB values of colours instead of pre-sets.
* While the controller is consistent, the launch direction still feels a bit off. Maybe a arcing line showing trajectory would be useful.
* Added, timer does not show an extra ‘0’ when in a single digit number, but numbers work just fine. Fine tuning this side of the script would be beneficial.

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  public class PlayerControllerScript : MonoBehaviour  {  // Variables needed  public Camera mainCam; // Uses main camera to get world position of mouse clicks later on  public Rigidbody playerCharacter; // reference to player rigidbody  public Text inputTrackingText; // refernce for where the 'inputTracker' int applies to on screen  public float power = 10f; // base number slings are calculated too  public bool canSling; // checks if its valid to sling  public LineTrajectoryScript lineTraj; // Calls other script here for line rendering  public float addedDirectionalForce; // Stores overall force of push  float addedDirectionalForceX; // Stores X value of mouseEndPoint and playerposition  float addedDirectionalForceY; // Stores Y value of mouseEndPoint and playerposition  public float minSlingDistance; // min required distance to draw line + allow the playr to launch themselves  public float maxSlingDistance; // max required distance to draw line + allow the playr to launch themselves  public bool requiredPower = true; // Connects to the 'EnergyMeterScript' to disable firing when not enough energy is present  public int inputTracker = 0; // Used for UI tracking of player inputs.  Vector3 currentMousePosition; // Actively tracked mouse position  Vector3 playerPosition; // tracks player position  Vector3 mouseEndPoint; //tracks where mouse is no longer held down  Vector3 launchDirection; // Impulse added to player character  public static PlayerControllerScript instance;  private void Awake()  {  instance = this;  }  private void Start()  {  // calls line trajectory script  lineTraj = GetComponent<LineTrajectoryScript>();  }  public void Update()  {  // Tracks current mouse position on every frame  currentMousePosition = mainCam.ScreenToWorldPoint(Input.mousePosition + new Vector3(0, 0, 12));  // Tracks players position, used for determining sling direction + line draw  playerPosition = playerCharacter.gameObject.transform.position;  // Min/Max launch distance code  // Minimum distance player can launch themselves too  if (Vector3.Distance(currentMousePosition, playerPosition) < minSlingDistance)  {  // If cursor is within minimum sling distance (i.e inside the player), then disable canSling  canSling = false;  }  else if (Vector3.Distance(currentMousePosition, playerPosition) > maxSlingDistance)  {  // If cursor is beyond the maximum sling distance, then disable canSling  canSling = false;  }  else  {  // If its within both the maxSlingDistance and the minSlingDistance, then set canSling to true  canSling = true;  }  // If its true, make the line blue  // If else, make the line red  if (canSling == true)  {  lineTraj.wantedLineRenderer.startColor = new Color(255, 255, 255);  lineTraj.wantedLineRenderer.endColor = new Color(0, 122, 255);  }  else  {  lineTraj.wantedLineRenderer.startColor = new Color(255, 255, 255);  lineTraj.wantedLineRenderer.endColor = new Color(255, 0, 0);  }  // While mouse is held down, render line between player position and currentMousePositions  if (Input.GetMouseButton(0))  {  lineTraj.RenderLine(playerPosition, currentMousePosition);  }  // Uses the players current position, and calculates the direction of the impulse  // by the position the mouse was before it was let go.  if (Input.GetMouseButtonUp(0))  {  // takes the end point the mouse cursor was at, used for calculating angle of impulse and distance  mouseEndPoint = mainCam.ScreenToWorldPoint(Input.mousePosition + new Vector3(0, 0, 12));  // minuses the mouseEndPoint by the playerPosition, then normalizes to return a smooth number that isn't a bloated vector (x and y seperate)  launchDirection = (mouseEndPoint - playerPosition).normalized;  addedDirectionalForceX = mouseEndPoint.x - playerPosition.x;  addedDirectionalForceY = mouseEndPoint.y - playerPosition.y;  // Removes negatives from force in X axis  if (addedDirectionalForceX <= 0)  {  addedDirectionalForceX = addedDirectionalForceX \* -1;  }  // Removes negatives from force in Y axis  if (addedDirectionalForceY <= 0)  {  addedDirectionalForceY = addedDirectionalForceY \* -1;  }  addedDirectionalForce = (addedDirectionalForceX + addedDirectionalForceY) \* power;  if (canSling == true && requiredPower == true)  {  playerCharacter.AddForce(launchDirection \* addedDirectionalForce, ForceMode.Impulse);  addedDirectionalForce = addedDirectionalForce \* 2;  EnergyMeterScript.instance.DrainPower(addedDirectionalForce);  inputTracker++;  TimerScript.instance.StartTimer();  }  // will end line even if canSling is set to false  lineTraj.EndLine();  // updates inputTracker UI element with amount of 'valid' inputs  inputTrackingText.text = ""+inputTracker;  }  }  } |

### Script 2 – Energy Meter

#### Script(s) generated

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  public class EnergyMeterScript : MonoBehaviour  {  public Slider energyMeter; // Calls slider  public Image fillAreaImage; // Calls fill area    Color validPowerAmount; // Sets fill area to blue  Color invalidPowerAmount; // Sets fill area to red  private float maxPower = 100; // Initiates max slider value  private float currentPower; // tracks current power value  private WaitForSeconds regenTick = new WaitForSeconds(0.1f); // custom timer set to 0.1 seconds  private Coroutine regen; // references a coroutine for regenning power  public static EnergyMeterScript instance; // instances the script for 'PlayerController' to use  // Prepares an instance in 'PlayerControllerScript' to subtract power from the slider  private void Awake()  {  instance = this;  }  // Sets current values that were ambiguous earlier.  private void Start()  {  currentPower = maxPower;  energyMeter.maxValue = maxPower;  energyMeter.value = maxPower;  validPowerAmount = new Color(0, 1, 255);  invalidPowerAmount = new Color(255, 0, 0);  }  // Function that takes 'addedDirectionalForce' from 'playerController' and uses it to remove a amount form the meter. Also has a bool which can prevent firing of the slingshot and colour changes  public void DrainPower(float amount)  {  if (currentPower - amount >= PlayerControllerScript.instance.addedDirectionalForce)  {  currentPower -= amount;  energyMeter.value = currentPower;  if (regen != null)  {  StopCoroutine(regen);  }  regen = StartCoroutine(RegenPower());  }  else  {  PlayerControllerScript.instance.requiredPower = false;  fillAreaImage.color = invalidPowerAmount;  }  }  // Resets 'requiredPower' to avoid a soft-lock, also resets fillArea colour  private void Update()  {  if (currentPower >= 30)  {  PlayerControllerScript.instance.requiredPower = true;  fillAreaImage.color = validPowerAmount;  }  }  // Initates a regeneration of the meter shortly after it has been consumed.  private IEnumerator RegenPower()  {  yield return new WaitForSeconds(0.1f);  while (currentPower < maxPower)  {  currentPower += maxPower / 80;  energyMeter.value = currentPower;  yield return regenTick;  }  regen = null;  }  } |

#### Functionality review

Similar to my player controller, the actual code seems to be significantly different than my prior pseudocode/flowcharts. A main example is the use of subroutines, which instead of just adding 5% every Fixed Update, it loops its function unless a yield prevents it. This is a lot neater than initially expected since it lowers overall CPU usage.

#### Implementation feedback notes

* Add some comments inside the functions detailing purpose.
* Neat and tidy, easy on their eyes
* Be careful when instancing other scripts within functions and if-statements.

#### Response to feedback notes

* Moved the comments above ‘DrainPower(float amount)’ into the function itself.
* While this may be the case, segmenting the variables clearer would be wanted in the future.
* With my limited understanding of calling other scripts, I built a majority of if-statements and major functions on top of this. And with such little time left, I can only prevent this in the future.

|  |  |
| --- | --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  public class EnergyMeterScript : MonoBehaviour  {  public Slider energyMeter; // Calls slider  public Image fillAreaImage; // Calls fill area    Color validPowerAmount; // Sets fill area to blue  Color invalidPowerAmount; // Sets fill area to red  private float maxPower = 100; // Initiates max slider value  private float currentPower; // tracks current power value  private WaitForSeconds regenTick = new WaitForSeconds(0.1f); // custom timer set to 0.1 seconds  private Coroutine regen; // references a coroutine for regenning power  public static EnergyMeterScript instance; // instances the script for 'PlayerController' to use  // Prepares an instance in 'PlayerControllerScript' to subtract power from the slider  private void Awake()  {  instance = this;  }  // Sets current values that were ambiguous earlier.  private void Start()  {  currentPower = maxPower;  energyMeter.maxValue = maxPower;  energyMeter.value = maxPower;  validPowerAmount = new Color(0, 1, 255);  invalidPowerAmount = new Color(255, 0, 0);  }  public void DrainPower(float amount)  {  // Function that takes 'addedDirectionalForce' from 'playerController' and uses it to remove a amount  // from the meter. Also has a bool which can prevent firing of the slingshot and colour changes  if (currentPower - amount >= PlayerControllerScript.instance.addedDirectionalForce)  {  currentPower -= amount;  energyMeter.value = currentPower;  if (regen != null)  {  StopCoroutine(regen);  }  regen = StartCoroutine(RegenPower());  }  else  {  PlayerControllerScript.instance.requiredPower = false;  fillAreaImage.color = invalidPowerAmount;  }  }  // Resets 'requiredPower' to avoid a soft-lock, also resets fillArea colour  private void Update()  {  if (currentPower >= 30)  {  PlayerControllerScript.instance.requiredPower = true;  fillAreaImage.color = validPowerAmount;  }  }  // Initates a regeneration of the meter shortly after it has been consumed.  private IEnumerator RegenPower()  {  yield return new WaitForSeconds(0.1f);  while (currentPower < maxPower)  {  currentPower += maxPower / 80;  energyMeter.value = currentPower;  yield return regenTick;  }  regen = null;  }  } |  |