James Moran

Amazing Archery! Design, Implementation and Testing

AE2 – Supplementary Report

Includes: Design, Implementation and Testing

Contents

[Design 2](#_Toc467655130)

[Basic Requirements 2](#_Toc467655131)

[See the Target (so the Player can aim at it) 2](#_Toc467655132)

[Allow the Player to Aim Their Bow (by moving their phone) 3](#_Toc467655133)

[Allow the Player to Adjust the Power of Each Shot (by dragging a touch input instrument (such as their finger), downwards across the screen) 4](#_Toc467655134)

[Player Class 6](#_Toc467655135)

[Implementation 7](#_Toc467655136)

[Basic Requirements 7](#_Toc467655137)

[See the Target (so the Player can aim at it) 7](#_Toc467655138)

[Allow the Player to Aim Their Bow (by moving their phone) 8](#_Toc467655139)

[Allow the Player to Adjust the Power of Each Shot (by dragging a touch input instrument (such as their finger), downwards across the screen) 9](#_Toc467655140)

[Testing 10](#_Toc467655141)

[Basic Requirements 10](#_Toc467655142)

# Design

## Basic Requirements

The ‘user stories’ for such requirements, are shown below:

* As a user, I want to be able to see the target down range, so I can aim at it
* As a user, I want to be able to aim my bow by moving my phone, so that aiming is simple and interactive.
* As a user, I want to be able to adjust the power of each shoot by pulling down the on the screen.
* As a user, I want to be able to hit the target and receive points for hitting it.
* As a user, I want to see my arrows flying in an arc so that is mimics real arrows in flight.
* As a user, I want to see a score board on the screen so that I can keep track of my score as I play.
* As a user, I want to be able to pause the game at any time.
* As a user, I want to be able to see a high score list so that I know how well I'm doing compared to my previous scores.
* As a user, I want to be able to see the student's name in the corner of the screen at all times so that I can give credit to the student.
* As a user, I want to be able to see a menu upon starting that game.
* As a user, I want to be able to pick from at least 2 shooting ranges.

For this set of ‘user stories’, the design for each ‘user story’ (from the top), is as follows:

### See the Target (so the Player can aim at it)

This project will use the Unity engine and the perspective for the Player; is a first-person perspective, looking straight towards the target.

A bitmap for the target (so that the Player can see it), will get shown in the scene, allowing the Player to see a target. A flowchart representation of such, is shown below:

I presume that I can simply drag and drop the bitmap into the scene.

The Player (whom would see the target whilst the game is playing), is not present in the game at this stage of design, their design is detailed in the next ‘user story’:

### Allow the Player to Aim Their Bow (by moving their phone)

For this ‘user story’; if the phone’s Acceleration has not altered since the last pass of this routine, the point of aim will not require adjustment to match the phone’s Acceleration, as this would not procure any results for the Player to see. This detail is delved into, below (as well as stepwise refinement, to achieve the intended level of pseudocode to allow the Player to aim by moving their phone):

1. Has the phone’s acceleration received adjustment since the last pass of this Aim Adjustment? If so; go to step 2, otherwise; go to step 3.
   1. Get phone acceleration.
   2. If the phone’s current acceleration is not equal to the phone’s last acceleration (of the last frame); go to step 2, otherwise; go to step 3.
2. Move the Player (and so, their perspective) in accordance with the acceleration.
   1. Get the difference in acceleration, for the phone’s Y-Axis.
   2. If this difference is positive; go to step 2.3, if the difference is negative; go to step 2.4, or if there is no difference; go to step 2.5.
   3. Move the crosshair upwards, then go to step 2.5.
   4. Move the crosshair downwards, then go to step 2.5.
   5. Get the difference in acceleration, for the phone’s X-Axis.
   6. If the difference is positive; go to step 2.7, if the difference is negative; go to step 2.8, or if there is no difference; go to step 2.9.
   7. Move the crosshair rightwards, then go to step 3.
   8. Move the crosshair leftwards, then go to step 3.
3. Store the phone’s current acceleration in a variable (LastPhoneAcceleration), for the next pass of this routine.
4. Aim Adjustment is now complete.

Of course, the Player’s aim is the point of aim getting adjusted, which will require the Player Character in game (with a 1st person perspective). They will also require a target reticle, to show their current point of aim.

The HUD for the Player Character, at this stage in design, appears as such:

### Allow the Player to Adjust the Power of Each Shot (by dragging a touch input instrument (such as their finger), downwards across the screen)

If the Player presses and holds their touch input instrument at a certain point on the screen, for greater than a certain quantity of time, prepare to modify their shot’s power level. Pseudocode to describe such, is shown below:

1. Get the current list of contact points with the screen.
2. If there are multiple contact points, within X quantity of time; go to step 3, otherwise; go to step 6.
   1. Get the current contact points of this touch screen.
   2. Check each value to determine if they are not null, if more than 2 values in this list are not null; go to step 3, otherwise; go to step 6.
3. Determine whether the Player is dragging upwards or downwards on the screen, if they are dragging upwards; go to step 4, otherwise; go to step 5.
   1. Check each point in this list.
   2. For each point in this list; if the z component of the contact point is greater than the initial contact point’s z component; go to step 4, otherwise; go to step 5.
4. Increase the power of the shot, then go to step 6.
5. Decrease the power of the shot, then go to step 6.
6. Finish Shot-Power adjustment here.

Now, to show the Player their power level, the HUD for them will require an update to reflect such, this is shown below:

### Hit the Target (and receive points for hitting it)

First, the Player must have fired an Arrow (the projectiles that can hit the target, are arrows, as confirmed by the next user story’s description), before even checking if a collision has occurred (using OnCollisionEnter(Collider other)). Then, assuming a collision has occurred; checking if this Arrow has collided with a target. If so; the Player will then score points, if not; the Player’s score will remain the same, as detailed below:

1. Has the Player fired an Arrow?
   1. Has the Player touched the ‘Fire’ button (UI Component), with their respective touch implement?
      1. Check Input.touches to see if the Player is interacting with the respective position, of the ‘Fire’ button.
2. Has a collision between this Arrow and another entity occurred?
   1. OnCollisionEnter(Collider other) is fired when this occurs
3. Is the other entity of this collision the Target?
   1. Check if the collision tag of the ‘other’ collider, is that of ‘Target’ (string)
4. Increase the Player’s score.
   1. Add X number of points to CurrentScore, by calling a function from a reference to the PlayerCharacter, within the Arrow class (where ‘X’ is the value that the Player’s score is increased by, every time they hit the Target (set as a constant value in the PlayerCharacterClass))

The design for the Player HUD, is now as follows:

Edit 1: Not initially considering other elements of the Player’s HUD at this stage, the Player’s HUD is actually as such:



### Player Class

At this point (as well as looking at the other basic requirements), the Player will require a class for:

* A representation of the bow they are using
* Retrieval and storage of their Current Shot Power Level
* Retrieval and storage of their Current Score
* Retrieval and storage of their Name
* A reference to their HUD
* Functions to modify the above properties

A behavior script (using C#, to represent the Player class), is to receive implementation, for the requirements listed above.

The first version of this class’s UML Class Diagram, is shown below, to the left:



Edit 1: This class now has an object pool, consisting of the arrows the Player can fire. For the time being, the Player can only have 10 arrows in flight, before collision (there are also alterations to other structures in the class, such as the method for handling shot power alteration now consisting of 2 methods, see left):

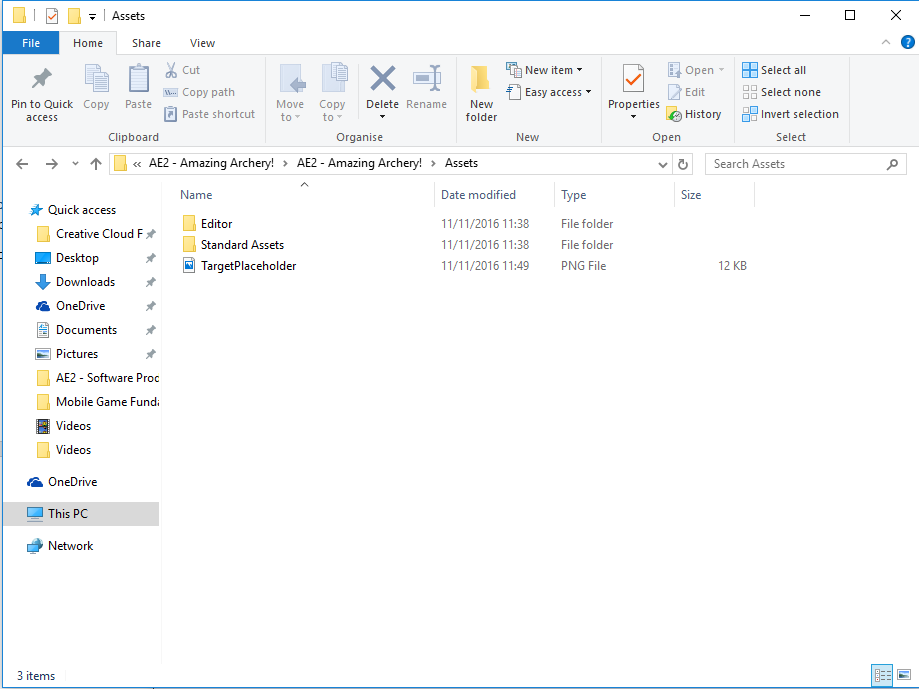
Edit 2: A pool of (Arrow) objects, is no longer used as the solution, there is now a constant value (const float MAX\_ARROWS\_IN\_PLAY = 10.0f), to limit the number of times instantiate is called to what was the object pool limit (when an object pool was going to receive utilisation, as the solution).

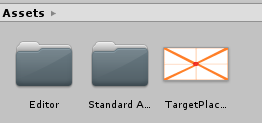
# Implementation

## Basic Requirements

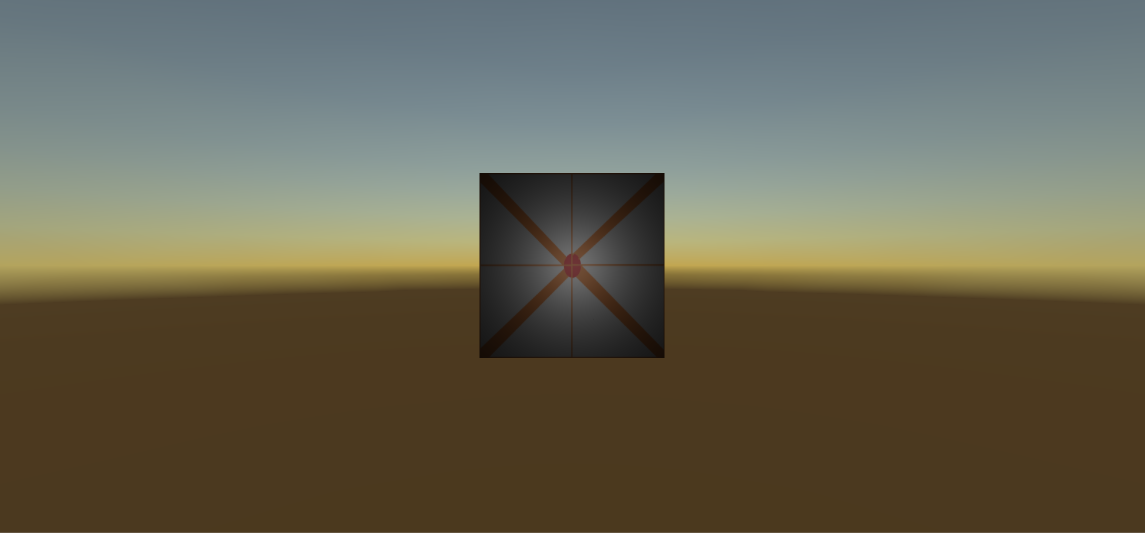
### See the Target (so the Player can aim at it)

First I will create the respective bitmap (so the Player can aim at it). I will use Microsoft Paint for this.

After this, I copy the respective bitmap into the assets folder (via the Windows Explorer):

I can also see this in the Unity’s asset browser:

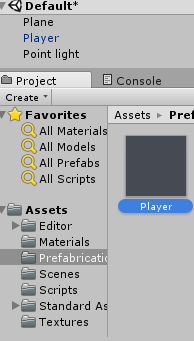
However, I am unable to simply drag and drop this bitmap into the scene (a 🚫 symbol appears), so I create a plane (by pressing the right-mouse button, whilst the focus is on the hierarchy panel of the Unity editor, then in respective context menu: 3D Object->Plane), then drag and drop the bitmap onto it, which is successful (but to see the object, a camera and a light were required, for a viewing perspective and a respective light source, after putting these into the scene in a similar manner to the Plane, as well as altering the transform of all these game objects…):



So, with the alterations to the initial implementation logic; it is now possible to see a target in the game.

### Allow the Player to Aim Their Bow (by moving their phone)

After creating additional folders to add structure to where assets are saved in the project, I dragged and dropped the Camera into the ‘Prefabrications’ folder, naming this prefabrication as ‘Player’ (see top of next page):

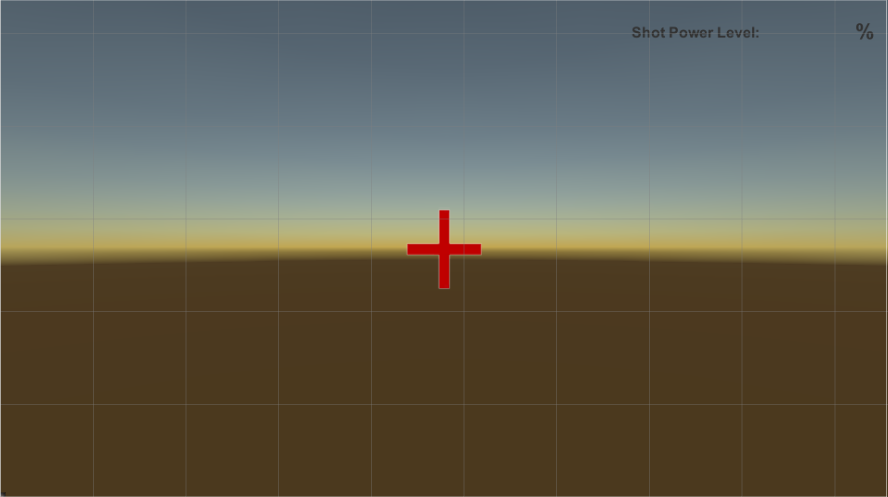
I then added a script to ‘Player’: A C# script known as ‘PlayerBehavior’.

I then implemented the respective functionality for this ‘user story’ (as per the design for this ‘user story’, from the ‘Design’ section), into this script.

Changes from the original design, are that of the following:

* AdjustAimPointForPhoneAcceleration would not require a parameter for the phone’s last or current acceleration (as it has class scope)
* In order to not have the game’s orientation rotate as per the phone’s oreintation, I set the logical orientation of the phone (via Screen.orientation) to the application’s default portrait orientation (with the use of ScreenOrientation.Potrait, where ScreenOrietation is an enum with 7 values, corresponding to logical accelerations of the respective device’s display)
* I thought that initially, implementation would utilise the phone’s gyroscope (gyro-sensor), I was later informed that this would not make sense, as indeed; the gyroscope handles the orientation of the device, whilst an accelerometer is capable for handling changes in velocity (such as rotating the phone at particular rate), so; I will use the accelerometer for implementation, instead of the gyroscope.
* Step 2 has received streamlining to a step without multiple sub-processes (calling one function, to handle movement of the Player)

### Allow the Player to Adjust the Power of Each Shot (by dragging a touch input instrument (such as their finger), downwards across the screen)

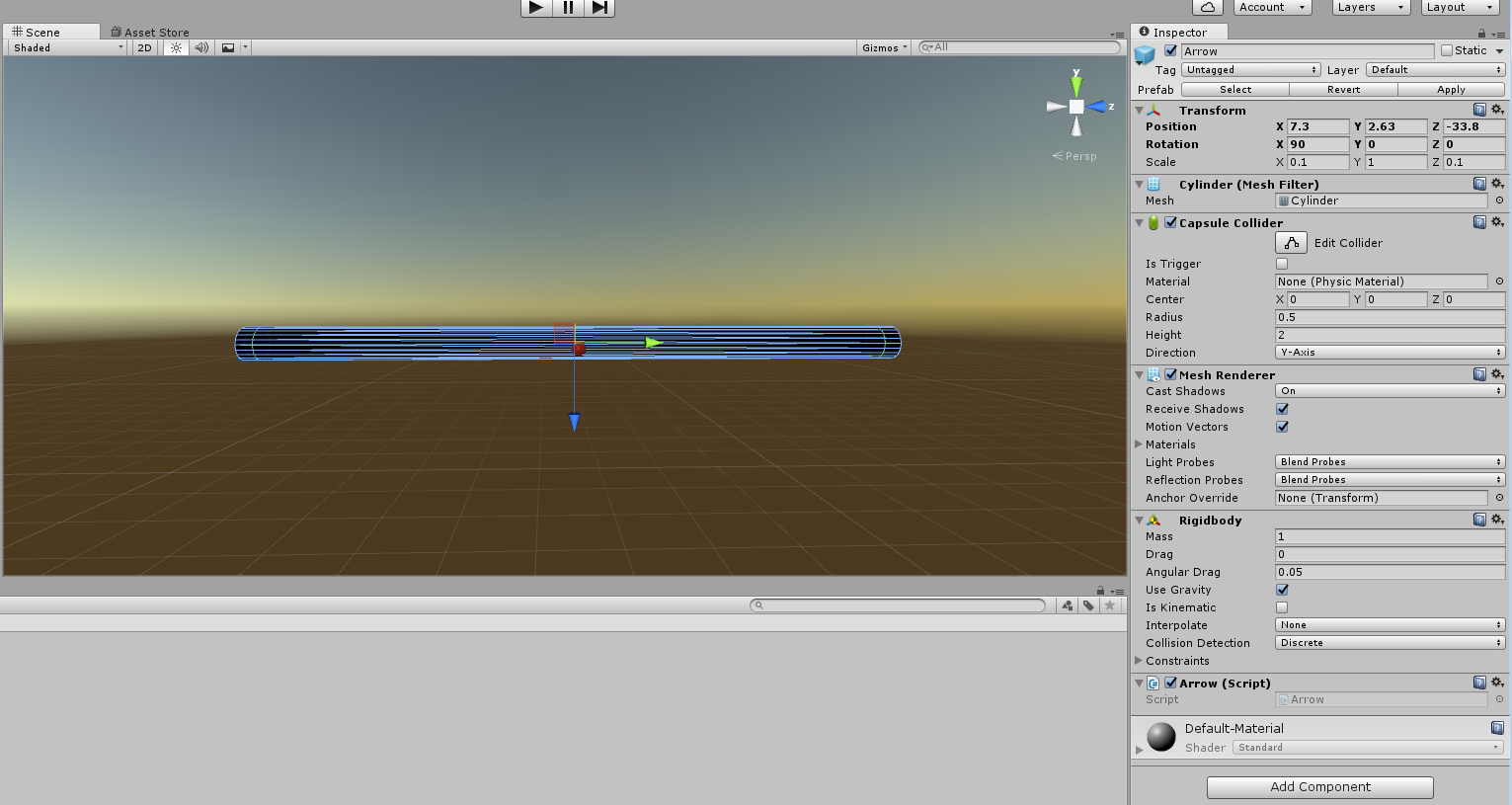
First, for showing the result of Power Level alteration; I create a UI object for both this and the Player’s crosshair (at this stage in design). This currently appears as shown below:

Now, the respective functionality for altering the power level, will receive implementation for ‘PlayerBehavior.cs’ (as per the design for this ‘user story’, from the ‘Design’ section).

Alterations to the original design; are that of the following:

* Checking the touchCount, instead of counting up the instances of touch input that are present is all that is required to check for if there is touch input
* Instead of checking if there are multiple touches, check to see that there is only one touch input (this frame), then check the phase of this input
* In checking the phase of this single touch input, the order required modification from what was thought in the initial order, as detailed in the pseudocode under the design section for this user story (
* Not all of the members of the respective touch input are required (only phase and location), so instead of storing a reference to this touch input, the functions for getting the required components, are called as and when the respective members of this touch input are required in the game loop

### Hit the Target (and Receive Points for Hitting It)

First; the Arrow prefabrication is created in the Unity Editor:

Then the collection to represent the object pool of arrows (for the Player), is declared and initialised: (NULL)

Edit 1: Instead of an object pool for arrows, the arrows are simply instantiated whenever they are required. Then destroyed whenever they hit a GameObject.

The HUD for the Player, has also received updates (shown below):

# Testing

## Basic Requirements

Black Box Testing is used here:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Description | Expected Results | Actual Results | Successful? |
| 1 | Precondition: Game is in play mode, within the Unity Editor.  Test: Upon starting the game, the Player can see the Target. | The Target is visible, at the centre point of the Player’s perspective. | The Target is visible, at the centre point of the Player’s perspective. | True |
| 2 | Precondition: Game is running on an Android device.  Test: The phone is rolled to the left (rotated negatively along the Y-Axis), relative to the phone, in the real world. | The Player’s point of view (aim), moves to the left of their current point of view. | The Player’s point of view (aim), moves to the left of their current point of view. | True |
| 3 | Precondition: Game is running on an Android device.  Test: The phone is rolled to the right (rotated positively along the Y-Axis), relative to the phone, in the real world. | The Player’s point of view (aim), moves to the right of their current point of view. | The Player’s point of view (aim), moves to the right of their current point of view. | True |
| 4 | Precondition: Game is running on an Android device.  Test: The phone is pitched upwards (rotated positively along the X-Axis), relative to the phone, in the real world. | The Player’s point of view (aim), moves upwards from their current point of view. | The Player’s point of view (aim), moves upwards from their current point of view. | True |
| 5 | Precondition: Game is running on an Android device.  Test: The phone is pitched downwards (rotated negatively along the X-Axis), relative to the phone, in the real world. | The Player’s point of view (aim), moves downwards from their current point of view. | The Player’s point of view (aim), moves downwards from their current point of view. | True |
| 6 | Precondition: The Game is running on an Android device.  Test: The Player touches the screen, then drags their touch implement downwards from their initial point of contact, before releasing their touch implement from contact with the screen. | The Player’s current power level Increases (if not already at 100). | The Player’s current power level increases (if not already at 100). | True |
| 7 | Precondition: The Game is running on an Android device.  Test: The Player touches the screen, then drags their touch implement downwards from their initial point of contact, before releasing their touch implement from contact with the screen. | The Player’s current power level decreases (if not already at 0). | The Player’s current power level decreases (if not already at 0). | True |