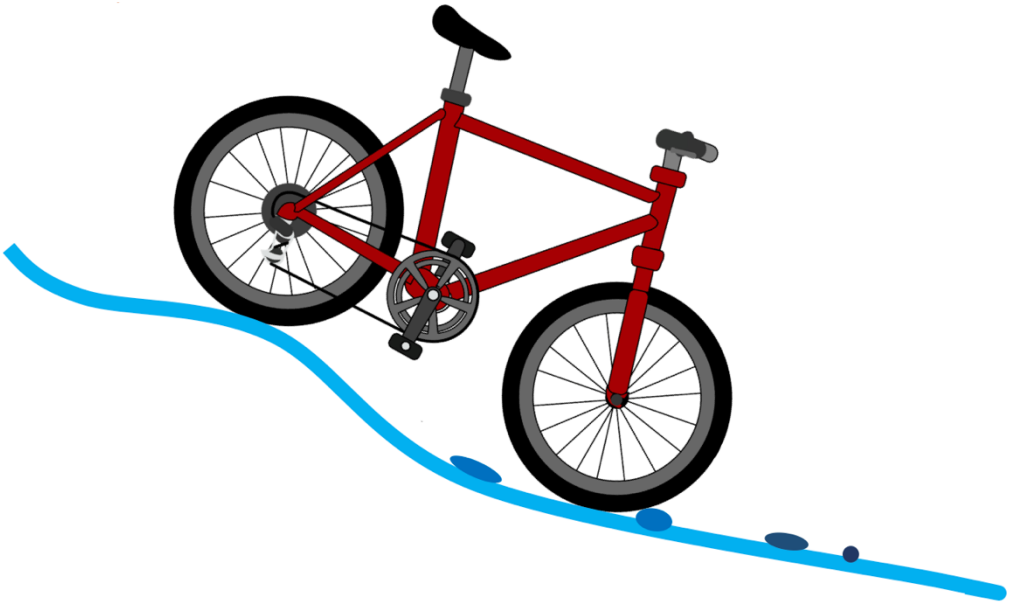
WheelCon

Platform Manual



By Quanying Liu & Ahkeel Mohideen

California Institute of Technology

Oct. 22, 2018

https://github.com/Doyle-Lab/WheelCon

Table of Contents

1.Installation 4

1.1 Preparation 4

1.2 Installation of WheelCon 4

2. Playing the Demo games 5

2.1 Fitt's law reaching game 6

2.2 Mountain bike task 7

2.2.1 Vision delay and action delay 8

2.2.2 Speed-accuracy tradeoffs in the plan control loop 9

2.2.3 Speed-accuracy tradeoffs in the reflex control loop 10

2.2.4 Bumps only, Trail only, and Bumps in the Trail 10

3. Model and data analysis 11

3.1.1 Modelling action delay in trail disturbance 11

3.1.2 Modelling action delay and quantization in trail disturbance 11

4. Low-level development 13

5. High-level development 15

5.1 Setting Up Unity Project for the First Time 15

5.2 Editing the Code 16

5.2.1 Description of Minor Scripts 17

5.2.2 Detailed Explanation of Major Scripts..... 18

5.3 Building the Game to an Executable..... 26

1.Installation

1.1 Preparation

The gaming platform need be run in windows 10 (not in Mac). Please make sure that you followed these steps in your testing computer.

a) Install Driver for your Steering Wheel. We have tested the platform on Logitech G27 and G29 gaming wheel and the THRUSTMASTER gaming wheel. The other gaming wheels are not guaranteed to be compatible with WheelCon.

   If you are using the Logitech G27, please download the driver here.

   If you are using the Logitech G29, please download the driver here.

   The THRUSTMASTER driver can be download at this link.

b) Get Logitech Steering Wheel SDK with this link. It will be used for testing/calibrating your wheel.

c) Plug in Steering Wheel and run their SteeringWheelDemo.exe to see if sensors can be read and wheel has a force feedback. If it doesn't work, please install the DirectX 9.0c August 2009 SDK or try DirectX Update.

1.2 Installation of WheelCon

Download the WheelCon from GitHub: (https://github.com/Doyle-Lab/WheelCon).

Unzip it, and run 'WheelCon.exe' at your testing computer with the Steering Wheel connected.

Start with the Demo game, then try to specify your own game. See details about the Demo games in **Chapter 2**.

In **Chapter 3**, we also provide an example of feedback control model for our demo game. We also provide MATLAB code for the output file generated from the Demo game. You have to install MATLAB to run it. However, the output file is simply in the 'txt' format, and you can write your own code to analyze them in any platform, such as R, or Python.

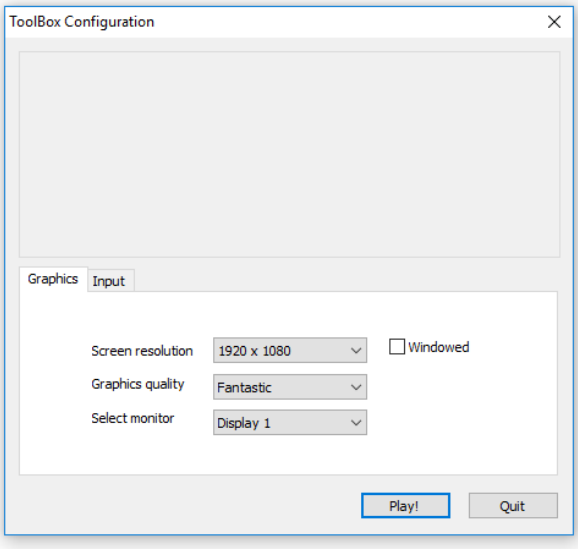
In **Chapter 4**, we show the lower-level development of new game using WheelCon. It is not needed to change the source code. You can simply generate new input files with specific format. We shared a simple example of MATLAB code to generate the input file for WheelCon.

In **Chapter 5**, we show the high-level development. You have to change the source code or write your own code. You need the basic knowledge of C# coding in Unity. Please install the visual studio for C# code and Unity for game development.

2. Playing the Demo games

We provide 6 demo games for you to start with.

Please start the platform by double click on ‘WheelCon.exe’. You will see the following screen:



Unselect the ‘Windowed’, and choose the correct Screen resolution for your monitor. Click ‘Play!’. You will see the main menu:

The Wheel Sensitivity (from 0 to 1) defines the sensitivity of wheel with low value meaning low sensitivity (you need a big angle of wheel to move the position).

Two tasks have been implemented in the platform: Fitts Law Task and Mountain Bike Task.

2.1 Fitt's law reaching game

When you choose the Fitts law game, it will go to the following gaming interface.

It mimics a traditional reaching task, which the player controls the steering wheel to the left / right, to reach a gray zone. The player should stay in the zone until the zone jumped to another area.

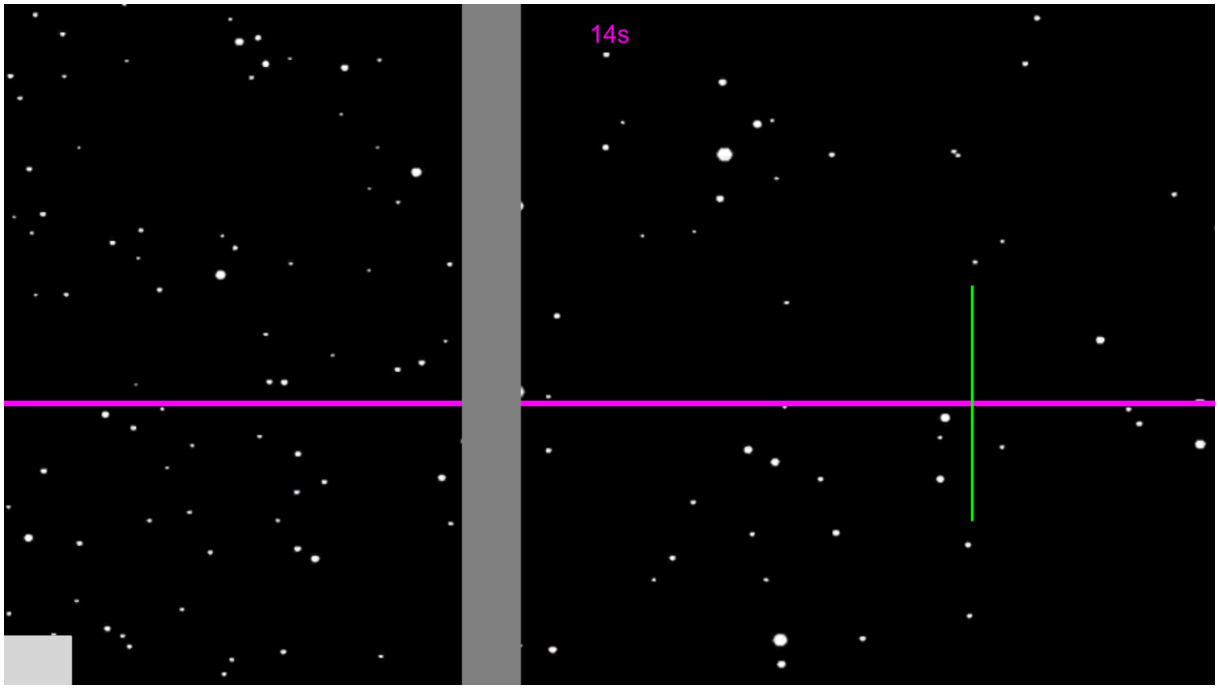
The Fitt's Law predicts that the time (T) required to rapidly move to a target area is a function of the ratio between the distance to the target and the width of the target. Let's define the distance to the center of the target (D) and the width of the target (W). T is a function of W and D:

$$\mathrm{T} = \mathrm{p} + \mathrm{q} \times log_2(\frac{2 \times W}{D})$$

where p and q are two parameters

See more details about the Fitt's Law in Wikipedia.

Please type the output file name (for saving the output file), select 'Demo Input Files\t_path_fitts_law.txt', and click 'Begin Game' to start the Fitt's Law task. You will see the following gaming interface:



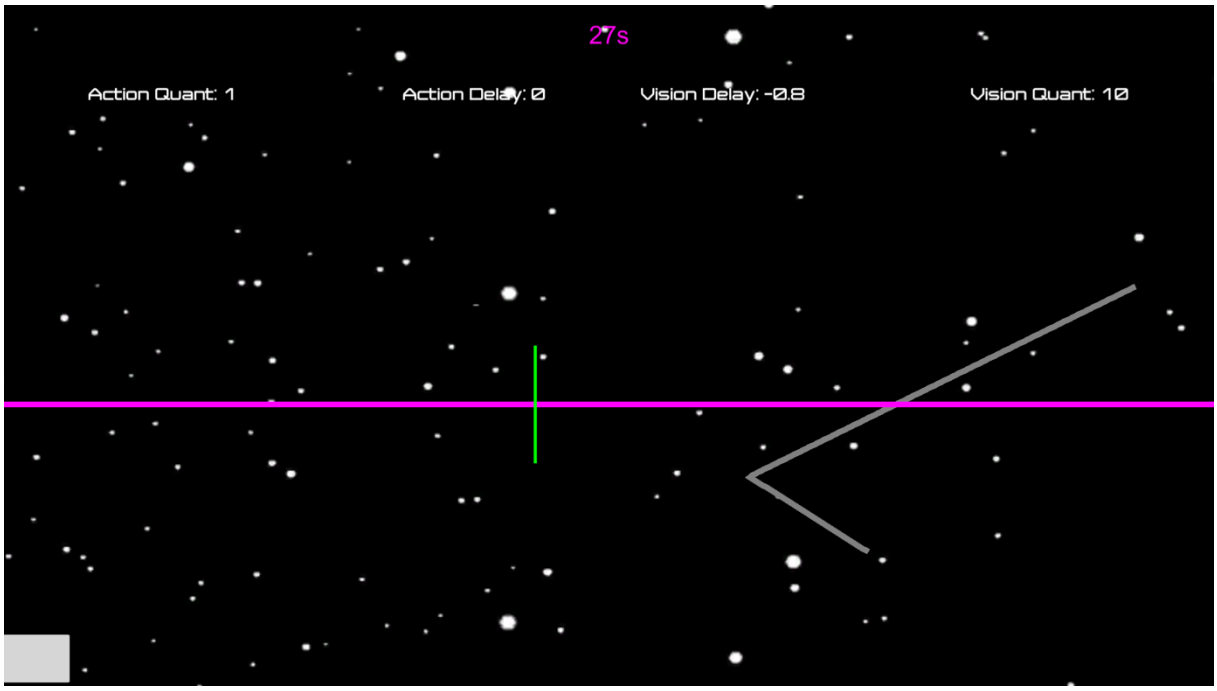
The gaming time is shown on top with purple text. The green line is your current position, and the grey zone is the desired zone to reach. The grey zone jumps each 2 second. When you see the gray zone in a new position, please steer the wheel to left or right to reach the zone and stay in the zone as quickly as possible.

The movement time is from the moment the grey zone jumping to another position to the time that the green bar staying in the grey zone.

2.2 Mountain bike task

When you choose ‘Mountain Bike Task’ in the main menu, it will go to the following gaming interface.

Please type the output file name, select 'Demo Input Files\setting_*.txt', and click 'Begin Game' to start the Mountain Bike task. You will see the following gaming interface:



The purple horizontal line indicates the present. The space above it indicates the future, and below it the past.

The grey line is the desired path, the green line is the player's current position. The task is to control the green line following the grey line with minimal error (the distance between the green line and grey line in the present time).

The quantization and delay for the visual input (negative delay means advance warning), and the quantization and delay for the action output are shown on top of the screen with white text. The gaming time is shown on top with purple text.

2.2.1 Vision delay and action delay

We start from the simplest game, with manipulating one parameter in the control loop: delay.

Please choose 'Demo Input Files\ setting_Vdelay_Adelay_worstcase.txt' to start the game.

In the game, we add either a visual delay or an action delay in the control loop. The visual delay is added via hiding the view of the trail. The action delay is added on the steering wheel.

We design a game with delay to test the impacts of delay. The sharp turns in the desired trajectory s(t) with the angle $\theta \in \{10, 20, \ldots, 80\}$ periodically arrive every 1.5 seconds. The disturbance r(t) = s(t+1)-s(t) with $||r||_\infty \leq 0.04$ unit (1 unit = 100 pixels). No advanced warning is added. Subjects therefore cannot see any future trajectory. To examine the effects of vision delay, we vary the

available history of trajectory in the visual feedback. At 0 delay, subjects cannot see the future trajectory, but they can see the full history of past trajectory. However, at 150ms, 300ms, 450ms, 600ms and 750ms of T_{vision} delay, they gradually lose the visual view of the past road. For the action delay test, an external delay is added to the steering wheel, but no vision delay. To be identical to the vision delay test, no advanced warning is provided. The 0ms action delay is the baseline setting; then 150ms, 300ms, 450ms, 600ms or 750ms of action delay, T_{action}, is added, all while keeping 0 vision delay so that the results are comparable between these two tests. Each setting in the game lasts for 30 seconds, so the total experiment is 6 minutes.

2.2.2 Speed-accuracy tradeoffs in the plan control loop

To test Speed-accuracy tradeoffs (SATs) in the higher-level plan layer, we design a game with visual advanced warning/delay and quantization. Please choose 'Demo Input Files\setting_SAT_worstcase_trail__ActQuant.txt' to start the game.

To test performance in the SATs with limited data rate (quantization) and advanced warning/time delay, we added either quantization in the actuator, or additional advanced warning/time delay in the visual input, or both. The quantization Q_{action} is put on the output from the steering wheel (the angle of the wheel), where the data rate, R_{action} is set to 1, 2, ..., 7 bits, respectively. The corresponding advanced warning/time delay, T_{vision}, is -800, -600, ..., 400ms, respectively. The negative value means advanced warning in the visual input, whereas the positive value means delay. For the T-R tradeoff settings, T = 200(R-5). Each setting lasted 30 seconds, so it takes 7*30*3=630 seconds in total (see Table 1). R_{vision} is 10 bits (maximum data rate for visual inputs) and T_{action} = 0 ms (no delay for the action output).

Table 1. Parameters setting in the SATs in the plan layer.

limited data rate	Gaming time (s)	0-30	30-60	60-90	90-120	120-150	150-180	180-210
	R_{action} (bits)	1	2	3	4	5	6	7
	T_{vision} (ms)	0	0	0	0	0	0	0
advanced warning/ delay	Gaming time (s)	210-240	240-270	270-300	300-330	330-360	360-390	390-420
	R_{action} (bits)	10	10	10	10	10	10	10
	T_{vision} (ms)	-800	-600	-400	-200	0	200	400
SATs	Gaming time (s)	420-450	450-480	480-510	510-540	540-570	570-600	600-630
	R_{action} (bits)	1	2	3	4	5	6	7
	T_{vision} (ms)	-800	-600	-400	-200	0	200	400

Please choose 'Demo Input Files\setting_SAT_worstcase_trail_VisQuant.txt' to start the game. In this game, we quantize the visual input. We only show the quantized gray line. The parameters are set the same as Table 1 but we vary R_{vision}. In this case, R_{action} is 10 bits (maximum data rate for action output) and T_{action} = 0 ms (no delay for the action output).

2.2.3 Speed-accuracy tradeoffs in the reflex control loop

To test SATs in the lower-level reflex layer, we design a game with bumps, and manipulate the delay and quantization in the action. Please choose 'Demo Input Files\ setting_SAT_Bump_worstcase.txt' to start the game.

The bump disturbance w(t) arrives every 2 seconds, with a maximum 100 units of torque. Similar to the setting for SATs in the advanced plan layer, we externally add either a quantizer with limited data rate R_{action}, or a delay T_{action}, or both in the steering wheel. The R_{action} is set to 1, 2, 3, 4 bits, respectively, and T_{action} is set to 0, 200, 400, 600ms, respectively. For the T-R tradeoff test, T = 200(R-1). Each setting lasted 30 seconds (in total 4*30*3=360 seconds). R_{vision} is 10 bits (maximum data rate for visual inputs) and T_{vision} = -1000 ms (800ms advance warning in the trail).

Table 2. Parameters setting in the SATs in the reflex layer.

limited data rate	Gaming time (s)	0-30	30-60	60-90	90-120
	R_{action} (bits)	1	2	3	4
	T_{action} (ms)	0	0	0	0
advanced warning/ delay	Gaming time (s)	120-150	150-180	180-210	210-240
	R_{action} (bits)	10	10	10	10
	T_{action} (ms)	0	200	400	600
SATs	Gaming time (s)	240-270	270-300	300-330	330-360
	R_{action} (bits)	1	2	3	4
	T_{action} (ms)	0	200	400	600

2.2.4 Bumps only, Trail only, and Bumps in the Trail

To be more realistic and natural, we developed a bump and trail dual-task game in the platform. Please choose 'Demo Input Files\ setting_Bump_Trail.txt' to start the game.

We either add bump disturbance w(t), or trail disturbance r(t), or bump and trail dual disturbance w(t) + r(t) in the game. To be noted, the w(t) and r(t) are independent. The bump effects were generated by a 0.5s constant torque to the steering wheel. We did not intentionally add external delays in the game (T_{action} = 0ms and T_{vision} = -1s), and we used the maximum data rate for both vision and action (R=10 bits).

Based on the feedback control model, we expect to see the sum of the error from the bump only task and from the trail only task is equivalent with the error from the bumps in the trail task. See more details about the model in the following chapter.

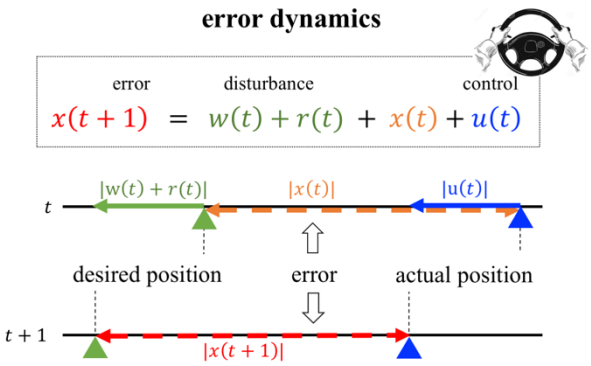
3. Model and data analysis

3.1 Feedback control model

We show a simplified feedback control model shown in Figure 4. The system dynamics is given by

$$x(t+1) = x(t) + w(t) + u(t) + r(t)$$

where x(t) is the error at time t, r(t) is the trail disturbance, w(t) is the bump disturbance, and u(t) is the control action.



3.1.1 Modelling action delay in trail disturbance

When there is a delay T in the action, and a trail disturbance r(t), we model the control action by

$$u(t+T) = \kappa\big(x(0:t), r(0:t), u(0:t+T-1)\big) .$$

The game starts with zero initial condition, i.e., x (0) = 0. The controller κ generates the control command u(t) using the full information on the histories of state, disturbance, and control input. The control command is executed with delay $T \geq 0$.

Sensorimotor control in the risk-aware setting motivates the use of L1 optimal control, and as such, our goal is to verify the following robust control problem

$$\inf_{\kappa} \sup_{\|r\|_\infty \leq 1} \|x\|_\infty .$$

This problem admits a simple and intuitive solution. The optimal cost is given by

$$\inf_{\kappa} \sup_{\|r\|_\infty \leq 1} \|x\|_\infty = T .$$

This optimal cost is achieved by the worst-case control policy $u(t+T) = -r(t)$, which yields

$$\inf_{\kappa} \sup_{\|r\|_\infty \leq 1} \|u\|_\infty = 1 .$$

3.1.2 Modelling action delay and quantization in trail disturbance

In the 'SATs in the plan control loop' game, the control action is generated by the following feedback loop with communication constrains,

$$u(t+T) = Q\big(\kappa_t\big(x(0:t), u(0:t-1)\big)\big) ,$$

where κ_t: $(\mathbb{R}^{t+1}, \mathbb{R}^t) \to \mathbb{R}$ is a controller, and Q: $\mathbb{R} \to S$ is a quantizer with data rate $R \geq 1$, i.e. S is a finite set of cardinality 2^R. Here, the net delay is composed of the internal delays in the

human sensorimotor feedback and the delays externally added. The disturbance r(t) is infinity-norm bound and without loss of generality, $\|r\|_\infty \leq 1$.

The worst-case state deviation is lower-bounded by

$$\sup_{\|r\|_\infty \leq 1} \|x\|_\infty \geq T + \frac{1}{2^R - 1} \ ,$$

and the minimum control effort is given by

$$\sup_{\|r\|_\infty \leq 1} \|u\|_\infty \geq \left(1 + \frac{1}{2^R - 1}\right)\left(1 - \frac{1}{2^R}\right) \ .$$

3.2 Data analysis

To analyze the data, we used MATLAB code. Please install MATLAB and run the demo 'Demo Input Files\setting_SAT_worstcase_trail__ActQuant.txt'. You will get an output file.

Please open 'Source Code\WheelCon_code_SAT_plan.m' with matlab, and adjust the variable 'folder' and 'file_names' to your folder and output filename.

Run the code, it will generate a figure with speed and accuracy tradeoffs in the performance.

Specifically, the raw data will be in the following variables.

```
M = dlmread(file_names,',',1,0);


% load the data
t = M(:,1);
trail = M(:,2);

bump = M(:,3);
quant_act = M(:,4);
delay_act = M(:,5);
delay_vis = M(:,6);
quant_vis = M(:,7);

error = M(:,8);
control = M(:,9);
```

This code if only for an example to analyze the txt output file in 'SATs in plan control loop task'. Feel free to write your own code to analyze the data.

4. Low-level development

For the low-level development, you only need write your own the input file. It is not necessary to know the source code with C# and unity.

The Input file needs to be setup in a specific way in order to get accurate experimenting. The file should be stored as a ".txt" file and all the information will be written in text. Each experimental variable is represented as an animation curve in the software. This means that at every time every experimental variable must be associated with a value. The resolution for the time is at minimum 10ms (0.01 seconds). Therefore, in the input file, all experimental variables must be valued at every time stamp, every 10 ms (or the desired time resolution). Each line of the text file should represent a single time stamp with appropriate variable values. The format of the line depends on the version of the game being played.

For the mountain task the format is:

Time, horizontal position of the trail, bump size, action data rate, action delay, vision delay, vision quantization

e.g. 0.01,6,10,-1,30,0.2

Please note the lack of spaces.

Each line will be read by the program, interpreted, and the necessary information inputted into their respective animation curves. The time is used as the input to the animation curve. There is no need for a line to signify the end of the game. The experiment will end as soon as the game time has exceeded the last declared time stamp in the input file. If the time in the last line of the input file reads 39.90, the game will end at 40 seconds.

In the mountain bike task in WheelCon, the forward speed has been set as 2.5 units / s. The limits for the horizontal position of the line is from -10 to 10, and the vertical position is from -4 to 6.

For the experimental variables, take the horizontal position of the trail as an example. The desired position of the trail ought to be set at every time stamp. In the example above, the trail will be at x(t) position when the current game time is at t (time= t). If you want the line to be straight and continue in the same direction, the trail position in next game time (at 0.01 seconds temporal resolution, time = t+0.01) should be

$$x(t+1) = (0.01s * 2.5 \text{ units} / s) * \tan(\theta),$$

where θ is the angle for the path direction that you designed, 0.01s * 2.5 units / s is the moving forward distance in 0.01 s.

This means to create a straight line the angle of the trail value should stay constant for as long as the experimenter wishes the line to be straight and continue in the same direction. If the

experimenter wants a straight line at an angle of 6 degrees for 10 seconds, there ought to be 1000 lines (assuming a time resolution of 10 ms) in the input file that read as such: time,6,⋯

All the other experimental values will work in this same manner. This is to say, do not expect the software to hold a certain experimental value constant until a change in value is desired or declared. The value of each and every experimental variable must be declared at every instant the game is in play. If no value is provided, the software will default to 0. If a value of 0 is desired, it is best to explicitly state it in the input file as ⋯,0,⋯

The horizontal position of the trail value represents the x position in the screen for the grey line.

The quantization level corresponds to the number of bits allowed for communication between wheel and computer. The wheel has a very fine resolution when measuring its current angle. It is worth noting that although the wheel can be rotated many times over, there is only 240 degrees of accepted angle range for the game's purposes. Adding quantization means that the wheel's angle is quantized into discrete regions. At a quantization level of 1, only 1 bit is allowed for communication between the wheel and computer. This means that there are only 21 angle values. In other words, all 240 degrees of movement are quantized to 2 possible angle values. These lie at the extremes of either maximum angle to the right (120 degrees) or maximum angle to the left (-120 degrees). With a quantization level of 1, if the wheel's current angle is less than or equal to 0, the software moves the player's position as if the wheel was at -120 degrees. If the wheel is at any angle greater than 0, the software moves the player as if the wheel was at 120 degrees. The reason the player is not allowed to go straight is because that would mean the player was capable of moving at -120, 0, or 120 degrees. This would be more than 2 options and so not capable of communicating with only 1 bit. With higher levels of quantization, player motion near perfectly reflects the angle of the wheel. The minimum level of quantization is 1. However, the way the software is written, there is technically no maximum level of quantization. The quantization level ought to be an integer value greater than 0. The software will run if decimal values are used but the number of quantization regions will be truncated to an integer within the software.

e.g.
quantization level of 2 → $2^2 = 4$ regions of motion → 240 degrees of motion is divided into 4 regions

quantization level of 3.5 → $2^{3.5} = 11.31$ regions of motion → 240 degrees of motion is divided into 11 regions (truncated)

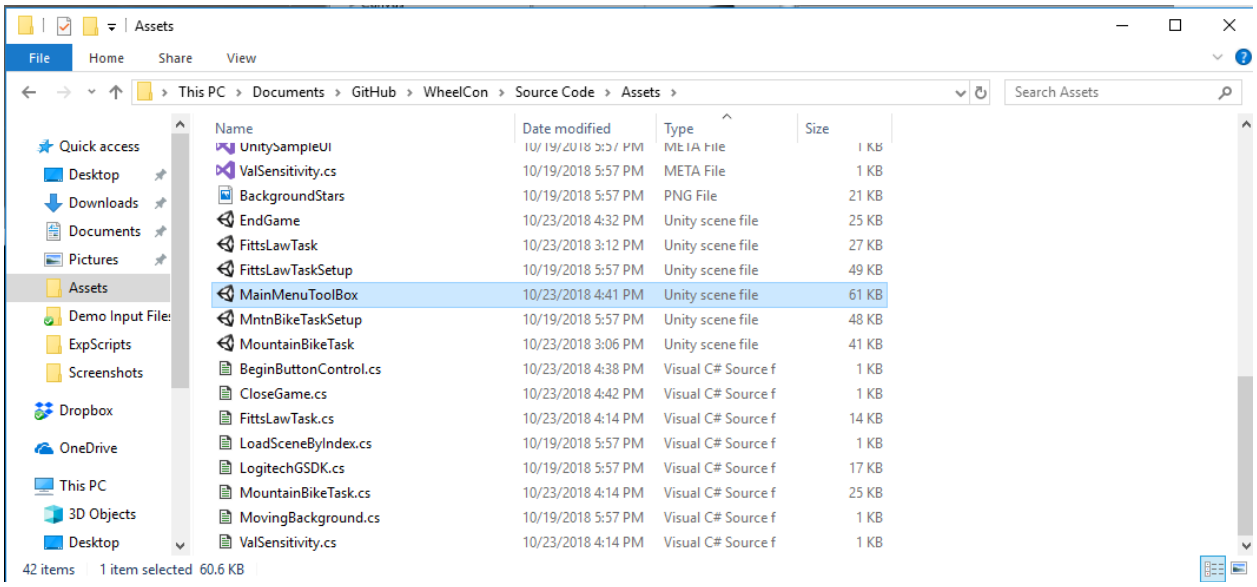
Please see an example MATLAB code for design the input file for a game at 'Source Code \ WheelCon Mntn Bike Trail Design Code.m', and feel free to write your own code in any language to design the input file.

5. High-level development

5.1 Setting Up Unity Project for the First Time

If necessary, the source code for the game can be altered as you see fit. To do so, necessary components such as Unity and Visual Studio need to be installed (see Installation). As you edit the program, the game can be played in the Unity scene-view. It is highly recommended that you watch a Unity tutorial before beginning. Several can be found on YouTube, but the Unity provides several tutorial projects, which can be viewed here.

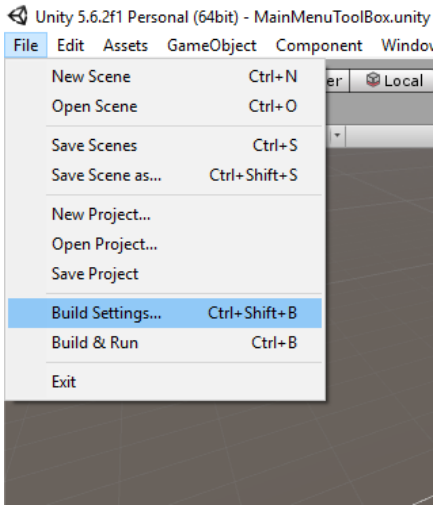
To begin editing our toolbox, open the Source Code folder. All the necessary source files are provided there, though Unity might generate some temporary files. Inside the Source Code folder, you should interact mostly with the Assets folder. Very rarely should the contents of the Project Settings folder be edited. If opening the project for the first time, double-click on the MainMenuToolBox.unity file to open the project.



The file should load as a scene in Unity.

Before editing anything, you need to add the other scenes to the build in a specific order. Navigate to File > Build Settings…

There you should see a list of the scenes in the current build. If it is your first time opening the project in Unity, likely there will be no scenes in the build. Each scene is given a number in the build, which can be viewed on the very right side of the dialog box on the line of the name of the scene.

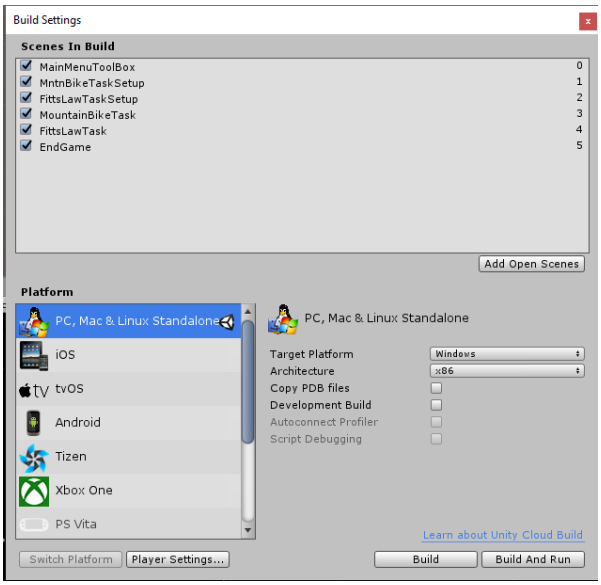


If MainMenuToolBox is not added to the build at 0, click the Add Open Scenes button. MainMenuToolBox should now be the only index in the build. An index of 0 means that Unity will launch that scene first when the program is started.

The remaining 5 scenes must be added to the build. However, the order at which they are added is important because otherwise the game will not function properly. To add the next scene, open the scene in Unity using the File Browser provided to the right of the scene-view. Then open the Build Settings dialog and press the Add Open Scenes button.

The order is as follows:

MainMenuToolBox – 0
MntnBikeTaskSetup – 1
FittsLawTaskSetup – 2
MountianBikeTask – 3
FittsLawTask – 4
EndGame – 5

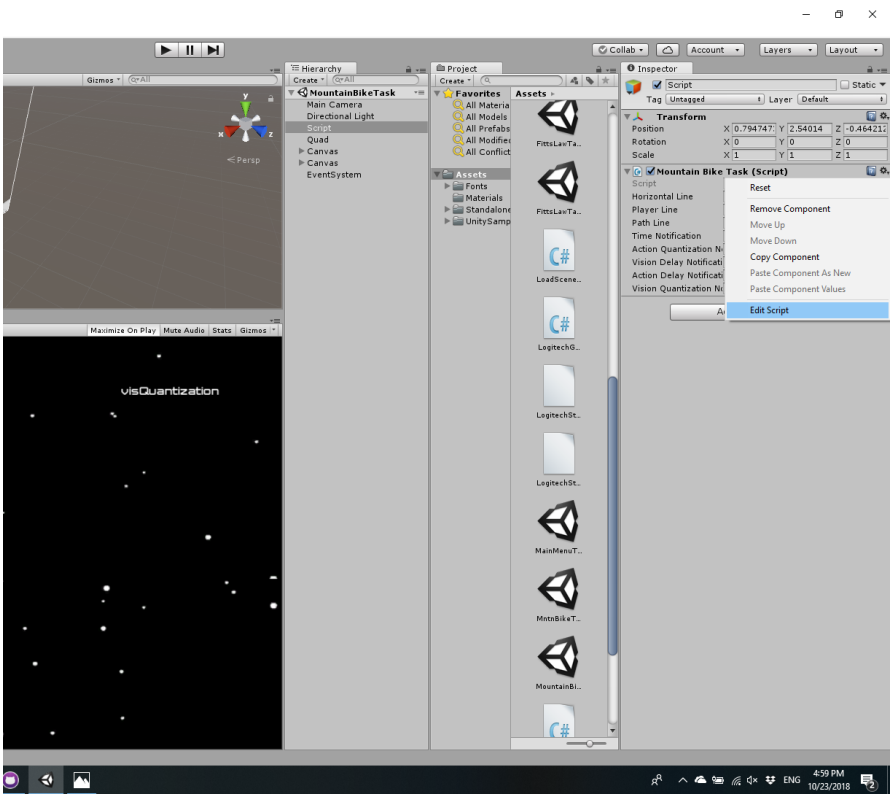


Once Completed, verify that your Build Settings appears as the picture to the right.

5.2 Editing the Code

Code for scenes are called Scripts in Unity. For simplicity, the code for the Fitt's Law Game and Mountain Bike Game have been condensed each to a single script per scene and have been named "Script". There are a few other scripts besides those two but they mainly are for navigation and handling menu settings (See Minor Scripts Descriptions for more information). The scripts for the tasks are presumably the most likely to be altered.

To edit the script for a scene, open the scene and find the “Script” game object. To edit the Fitt’s Law Game, open the FittsLawTask scene. To edit the Mountain Bike Game, open the MountainBikeTask scene. Click on the script in the Game Object Hierarchy. In the Inspector window, click on the gear to the right of the script and select edit. The code should open in Visual Studio. See the picture to the right.



The code is heavily commented with explanations at every section. For further explanation see the Description of Minor Scripts or Explanation of Major Scripts sections of this manual.

5.2.1 Description of Minor Scripts

There are 8 scripts present in the entire project. Each of these is fundamental to the function of the game. Below you will find descriptions of each of the classes, so that you may understand better how they fit into the game.

- BeginButtonControl.cs

  This class controls the GUI of the Task Setup scenes for the game by enabling the “Begin” button for each task. Only if an output file name and input txt file are provided, does the “Begin” button enable itself, so the user may begin the task.

- LoadSceneByIndex.cs

  This class has only one method. Its sole purpose to load a scene given its integer index. This integer corresponds to the number the scene is given in the build settings. Thus, it is important that the scenes are in the correct order in the build settings. This class and its method are used several times throughout the project to hop between scenes.

- LogitechGSDK.cs

  This class holds all the properties for the wheel. It is highly recommended that this class never be edited.

- MovingBackground.cs

  This class controls the background image for the Mountain Bike Task. To give the impression of forward motion, the background moves toward the bottom of the screen. The rate at which this happens, can be edited in this class. The speed at which the background moves is set to 2.5 units/sec. This is calibrated with the forward speed of the trail, which is also 2.5 units/sec.

- ValSensitivity.cs

  This class controls the calibration between the sensitivity slider and the value shown in the input text field beside it. This is so that if the slider value changes, the value shown reflects the position of the slider and vice-versa. It also holds the value of the sensitivity, so it can read later by the program. The sensitivity is proportional to the horizontal speed of the player cursor.

- CloseGame.cs

  This class quits the application when its sole method is called. This class and its method are called when the "Exit Game" button is pressed. It has no other utility.

5.2.2 Explanation of Major Scripts

Using snippets of the code, this section will attempt to provide as much detail about the construction of the scripts responsible for the Fitt's Law Task and Mountain Bike Task. Major methods will be explained in this section, with information provided on how they fit into the overall function of the program. Some proprietary understanding of Unity scripts and basic C# libraries is necessary, though some explanation will be provided.

Each of the class variables is accompanied by a comment explaining the meaning of the value held. Therefore, no further explanation of variables will be provided for class variables. If the purpose or type of variable is unknown, check online in the Unity or C# API's.

Explanation of Unity Script Methods

```
void Start() {
…
}
```

The Start() method is called at the creation of the game object the script is a part of. Most likely, this will be the creation of the scene. Therefore, the code inside this method is the first to run. Most importantly, the Animation Curves are all instantiated inside this method. In addition, GUI elements are initialized.

```
void Update() {
…
}
```

The Update() method is the continuous loop function of the Unity game. It is called as frequently as possible. In short, when the program gets to the bottom of the method it circles back to the top in a continuous loop. Therefore, all the code must somehow be referenced through the Update method.

Fitt's Law Task Explanation

```
public void LoadTestParameters(ref AnimationCurve distance, ref AnimationCurve
width)
{
    distance = new AnimationCurve();
    distance.preWrapMode = WrapMode.ClampForever;
    distance.postWrapMode = WrapMode.ClampForever;
    distance.AddKey(0, -1);


    width = new AnimationCurve();
    width.preWrapMode = WrapMode.ClampForever;
    width.postWrapMode = WrapMode.ClampForever;
    width.AddKey(0, -1);

    try
    {
        string line;

        StreamReader theReader = new StreamReader(inputFilePath,
System.Text.Encoding.Default);

        using (theReader)
        {

            float time;
            float distanceVal;
            float widthVal;


            do
            {
```

```
                line = theReader.ReadLine();

                if (line != null)
                {
                    string[] entries = line.Split(',');


                    if (entries.Length > 0)
                    {
                        time = Convert.ToSingle(entries[0]);
                        widthVal = Convert.ToSingle(entries[1]);
                        distanceVal = Convert.ToSingle(entries[2]);

                        width.AddKey(time, widthVal);
                        distance.AddKey(time, distanceVal);

                        gameLength = (int)time + 1;

                    }
                }
            }
            while (line != null);
            /* Done reading, close the reader and return true to broadcast success
*/
            theReader.Close();
            return;
        }
    }
    /* If anything broke in the try block, we throw an exception with information
     * on what didn't work */
    catch (Exception e)
    {
        Debug.Log("ERROR: " + e.Message);
        return;
    }


}
```

Called during the Setup(), this method loads in all the information from the input text file into distinct Animation Curves. To do this, each line is read from the text file and separated into separate numbers looking at the commas as the breaks between values. Then for each read value, a key (the first number of the line – the time) and value are entered into the Animation Curve, which will be used to hold and communicate the information throughout the rest of the program. This is the same process used in the Mountain Bike Task for reading the input text file.

```
public void wheelPropSetup(ref LogitechGSDK.LogiControllerPropertiesData prop)
{
    /* set up wheel to allow full range of motion and set gains
     * input is pointer to struct to hold wheel properties */

    /* allow full range of motion */
    prop.wheelRange = 900;

    /* set gains for the steering wheel: */
    prop.forceEnable = true;
```

```
    prop.overallGain = 80;
    prop.springGain = 80;

}
```

This method sets up the properties for the wheel, during the Start() process. It isn't recommended that this method be tinkered with unless specific settings are provided for the wheel you are employing.

```
/* update width of goal zone with info already loaded from text file */
goalLineWidth = widths.Evaluate(gameTime);
goalLine.SetWidth(goalLineWidth, goalLineWidth);
goalLineXPos = distances.Evaluate(gameTime);

goalLine.SetPositions(new Vector3[2] { new Vector3(goalLineXPos, -goalLineHeight,
0), new Vector3(goalLineXPos, goalLineHeight, 0) });
```

On every update, the width and distance from the center values of the goal zone are evaluated from their respective Animation Curves. Then these settings are applied. So the width of the goal zone becomes the width of the line renderer used as the goal zone. And, the position of the goal zone is set to the distance value.

```
playerLineXPos = sensitivity * NormalizedAngle;

/* keep the player position from going off the screen */
if (playerLineXPos < XminScreen)
{
    playerLineXPos = (float)XminScreen;
}
else if (playerLineXPos > XmaxScreen)
{
    playerLineXPos = (float)XmaxScreen;
}

/* set position of the player line */
playerLine.SetPositions(new Vector3[2] { new Vector3(playerLineXPos, -
playerLineHeight, 0), new Vector3(playerLineXPos, playerLineHeight, 0) });
```

On every update, the player's position is set to the angle of the wheel multiplied by a constant which is the sensitivity. This means that if the wheel stays at 60 degrees, the players position will stay the same. It doesn't keep moving right. The angle of the wheel proportionally reflects the position on the screen, in contrast to the Mountain Bike Task. Saturation occurs so that the player does not move off the screen even with greater angles.

```
public void saveScoreFile(string[] data)
{
    /* Method responsible for outputting
    * saved positions and data for error
    * calculation later. Data to be written
    * in the order of:
    * time, player position, width, distance (Goal Position)
    *
    * FILES SAVED IN "Executable & Output Files/FittsLawData"
```

```
    */

    string outputFileName = RecordFileName + "_" + gameDate + ".txt";
    string outputFilePath = saveFilePath + "/FittsLawData/";


    /* check if directory exists & create if not */
    if (!Directory.Exists(outputFilePath))
    {
        Directory.CreateDirectory(outputFilePath);
    }

    /* Save Score: */
    outputFilePath += outputFileName;

    /* If file doesn't exist yet, create it */
    if (!File.Exists(outputFilePath))
    {
        File.WriteAllText(outputFilePath, "");
    }
    File.WriteAllLines(outputFilePath, data);

}
```

Error data is compiled on every call of the Update() method. This data is compiled into a list rather than written to a file each call to increase the speed of the game. At the end of the game, before exiting to the End Game screen, this method is called and all the error data is written to the output text file in the format specified in this manual.

Mountain Game Task Explanation

```
/* add recorded angle to list with time at which that angle occured */
wheelAngles.Add(new float[] { gameTime, normalizedAngle });

/* find angle at time with appropriate delay & remove unnecessary from list */
for (int i = wheelAngles.Count - 1; i >= 0; i--)
{

    if (gameTime - wheelAngles[i][0] > actionDelay)
    {
        quantizedAngle = QuantizeAngle(wheelAngles[i][1], actionQuantizationLevel);
        wheelAngles.RemoveRange(0, i);
    }

}
```

The above section of code enables the Action Delay function of the game. On every update, a 2-element array containing the current game time and current angle of the wheel is added to a list called wheelAngles. Therefore, wheelAngles contains a list of the angle the user put the wheel at and the time the user did that.

If there was an action delay of 0.5 sec, then that would mean that there is a 0.5 sec delay between when the user sets the wheel to a certain angle and the player position reflecting that angle. In the wheelAngles list, because each angle is accompanied with a time stamp, it is easy to instantiate this delay. The program loops through the list finding the element where the difference between

the time stamp and the current game time becomes greater than the action delay. The condition cannot be changed to find when the difference becomes equal to the action delay because lag in the computer would ensure that you almost never find a difference exactly equal to the delay specified. The computer takes the index found, gets the accompanying wheel angle, quantizes it and sets it equal to the quantized angel, which changes the player position.

```
playerLineXPos = playerLineXPos + sensitivity * quantizedAngle;

/* create force on the wheel from input file */
LogitechGSDK.LogiPlayConstantForce(deviceIdx, (int)bumpSize);

/* keep the player position from going off the screen */
if (playerLineXPos < XminScreen)
{
    playerLineXPos = XminScreen;
}
else if (playerLineXPos > XmaxScreen)
{
    playerLineXPos = XmaxScreen;
}

/* set position of the player line */
playerLine.SetPositions(new Vector3[2] { new Vector3(playerLineXPos, -
playerLineHeight, 0), new Vector3(playerLineXPos, playerLineHeight, 0) });
```

The player's position is set in this code. It is equal to the sensitivity multiplied by the quantizedAngle found from the code above added to the position before. This means that the position is additive and more like a riding a real bicycle. For example, even if the angle of the wheel stays at 1 degree, eventually the player will reach all the way to the right side, unlike the Fitt's Law Task. However, the both share the same saturation feature to keep players from going off the edges of the screen. Lastly, player's position line is updated with the new calculated position.

```
public float QuantizeAngle(float wheelAngle, int quantLevel)
{

    /* quantization level is the number of bits available for quantization of input
*/
    int numLevels = (int)Math.Pow(2, quantLevel);
    float angleDifference = totalAngleRange / numLevels;

    /* if wheelAngle is greater than largest angle in range
     * return max value of angle range */
    if (wheelAngle > totalAngleRange / 2)
    {
        return totalAngleRange / 2;
    }
    else if (wheelAngle < -totalAngleRange / 2)
    {
        return -totalAngleRange / 2;
    }
    /* else look which quantized angle value the wheel angle is closest too */
    else
    {
        /* loop through quantized levels */
        for (float quantizedAngle = 0; quantizedAngle <= (totalAngleRange / 2);
quantizedAngle += angleDifference)
        {
```

```
            /* skip 0 because can't add 0 as possible quanitize level */
            if (quantizedAngle != 0)
            {
                if (Math.Abs(wheelAngle) <= quantizedAngle)
                {
                    /* return negative of quantized angle
                     * if wheel angle is negative */
                    if (wheelAngle < 0)
                    {
                        return quantizedAngle * -1;
                    }
                    else
                        return quantizedAngle;

                }
            }
        }
    }

    /* this option should never run */
    return 0;
}
```

Angle Quantization is enabled through this section of the code. Given an angle and the quantization level, the code returns the quantized value of the angle. There are 180 degrees of available motion in game. Any angle value beyond, is saturated to the max. The function relies on symmetry. The quantized angle becomes the ceiling value of the wheel angle. So with a quantization level of 2. The possible angles are -90, -45, 45, 90. The absolute values are compared. Ceiling value means that a wheel angle of 46 would be quantized to 90, because it is greater than 45. Likewise, an angle of -46 would be quantized to 90 then a multiplier of -1 would be added to make the quantized angel -90. Never will there be a quantized angle of 0, because it is breaks the symmetry and requires an extra bit. 0 is therefore used as an error handler. If the quantization code breaks, 0 becomes the quantized angle for easy debugging.

```
public float QuantizeTrailPosition(float trailPosition, int quantLevel)
{
    /* quantize horizontal axis into lanes of equal and discrete
     * width. Then report trail position as the center of the lane
     * which the trail position value from the file is closest to */

    /* find number of lanes */
    int numLevels = (int)Math.Pow(2, quantLevel);

    /* from that find width of each lane */
    float deltaX = (float)(XmaxScreen - XminScreen) / ((float)numLevels);

    /* keep trail from going off the screen */
    if (trailPosition > XmaxScreen)
    {
        return XmaxScreen;
    }
    if (trailPosition < XminScreen)
    {
        return XminScreen;
    }

    /* find the truncated number of lanes trail position from file is from
```

```
     * leftmost of the screen */
    int leftBlock = (int)Math.Floor((trailPosition - XminScreen) / deltaX);

    /* if it is the edge of right most block, then subtract one so
     * the position is made to the center of the rightmost */
    if (leftBlock == numLevels)
    {
        leftBlock = numLevels - 1;
    }

    /* start from the left of the screen, add the number of lanes
     * times the width of each lane, plus one half a lane to put position
     * in the center of that lane */
    return XminScreen + leftBlock * deltaX + deltaX / 2f;
}
```

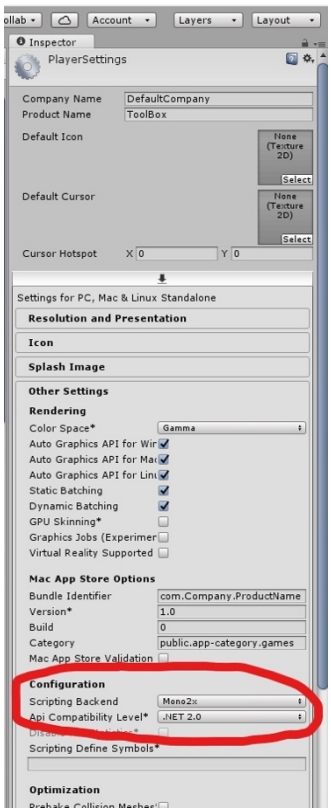
Quantization of the trail position enables vision quantization. Imagine the screen divided into lanes. The trail can only exist in the center of the lane, not in between lanes or the side. If the trail changes lanes, it does so instantly at 90 degree angles. This is, in essence, how vision quantization works. The screen is split into a number of lanes exponential to the quantization level. The trail is put at the center of the lane to which it is closest. When the value of the trail position crosses over into the next lane the trail transports to the center of that lane. Saturation of the trail occurs at this level, so that the trail may not exit the visible screen. To decide which lane, the position of the trail is subtracted from the left most position on the screen and the difference is divided by the width of each trail. The number is truncated to an integer. Truncating the value creates the quantization.

```
public float QuantizeTrailWidth(int quantLevel)
{
    /* set the width of the trail in accordance to
     * width of the lane from quantized trail */

    int numLevels = (int)Math.Pow(2, quantLevel);

    float width = (float)(XmaxScreen - XminScreen) / (float)numLevels;

    /* set min and max width */
    if (width < 0.1f)
    {
        return 0.1f;
    }
    else if (width > 1f)
    {
        return 1f;
    }
    return width;
}
```

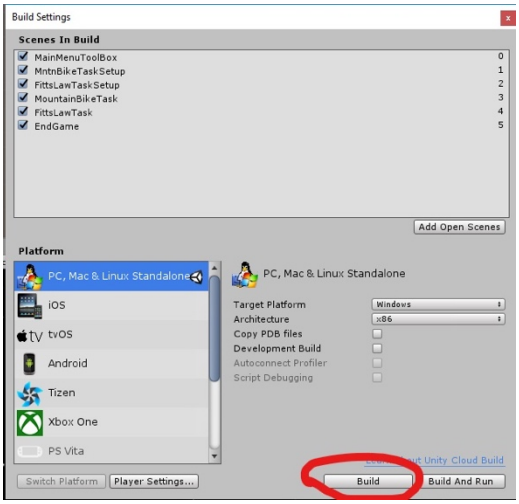
For visual quantization, it is important that not only the location of the trail is quantized but the width as well. The width should be equivalent to the size of the "lanes" above. The width is saturated however, because there would be a point when the trail could no longer be seen. The width is saturated between 0.1 and 1 units. Here, the same code used to find the deltaX variable above is used to find the width of the trail before saturation.

5.3 Building the Game to an Executable

There are a few checks that need to be run before building the game. Exporting to an .exe can be tricky because it is very possible that the game works in the scene view but will break when exported to an executable. There are log files written to in the Data folder created along with the executable upon build. These logs provide some information but are not necessarily the best. Therefore, it is important the possibility of error be minimized prior to export. To do so, follow these steps.



If employing the StandaloneFileBrowser package provided in the Assets folder, it is necessary that the Player Settings be altered. Navigate to File > Build Settings… Then press the Player Settings… button in the lower left corner of the dialog box. The player settings inspector window should open. Find the API Compatibility Level setting under the Configuration header. Set the API Compatibility Level to ".NET 2.0". Make sure it is **not** set to ".NET 2.0 Subset". See the picture to the right.



To build the game, navigate to File > Build Settings… Verify that all the scenes are added in the correct order. Make sure that the Architecture is set to x86_64 and **not** x86. Now, it is possible to build the game as a Developmental Build. This allows for some debugging abilities, so that the game will provide an error message if a problem occurs instead of just crashing. If all the Build Settings are done properly, press the Build button.