A1 Report

Raymond Dee – 100578667

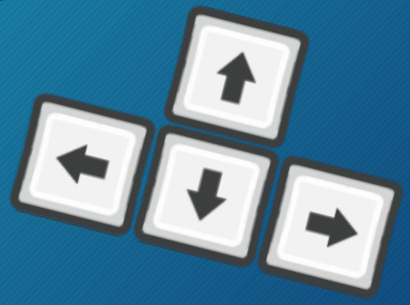
Cameron van Velzen - 100591663

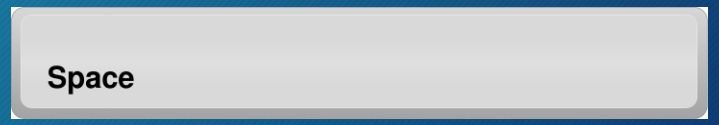
The Game

The basics of our game executable

The Controls

- By default, the car is driven using the WSAD controls
- Alternatively, when the controls are switched, it's driven with the Up/Down/Left/Right Arrow keys
- Finally, you can use the Space Bar to shoot bullets, and the Left Mouse Button to click onscreen buttons

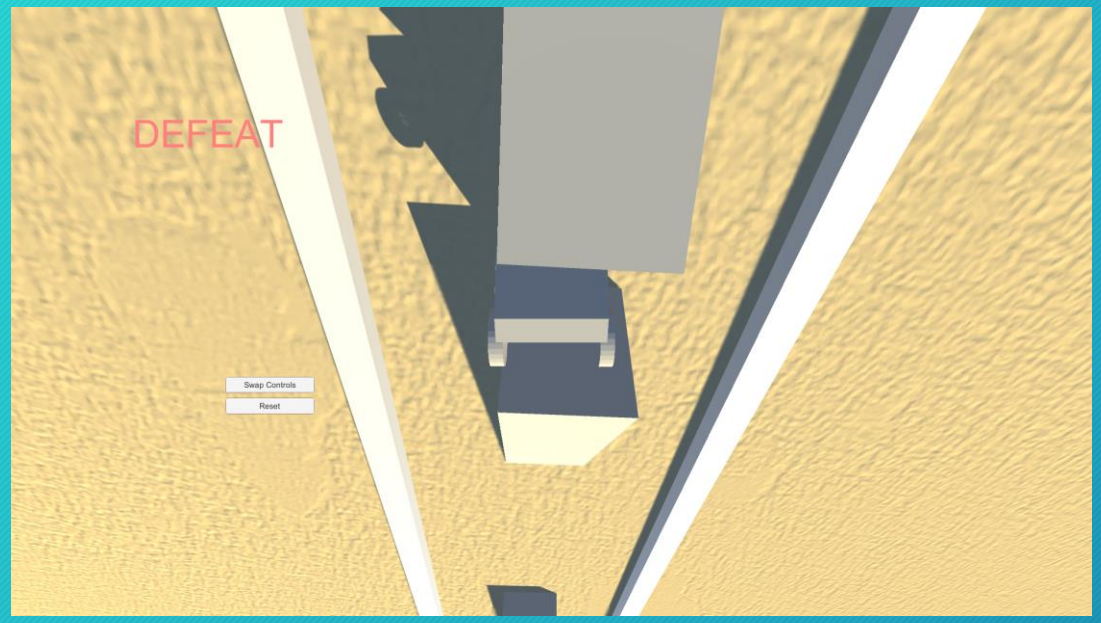




The Objective

- The objective of the game is simple, clear all the obstacles to make it through the finish line!

The Objective



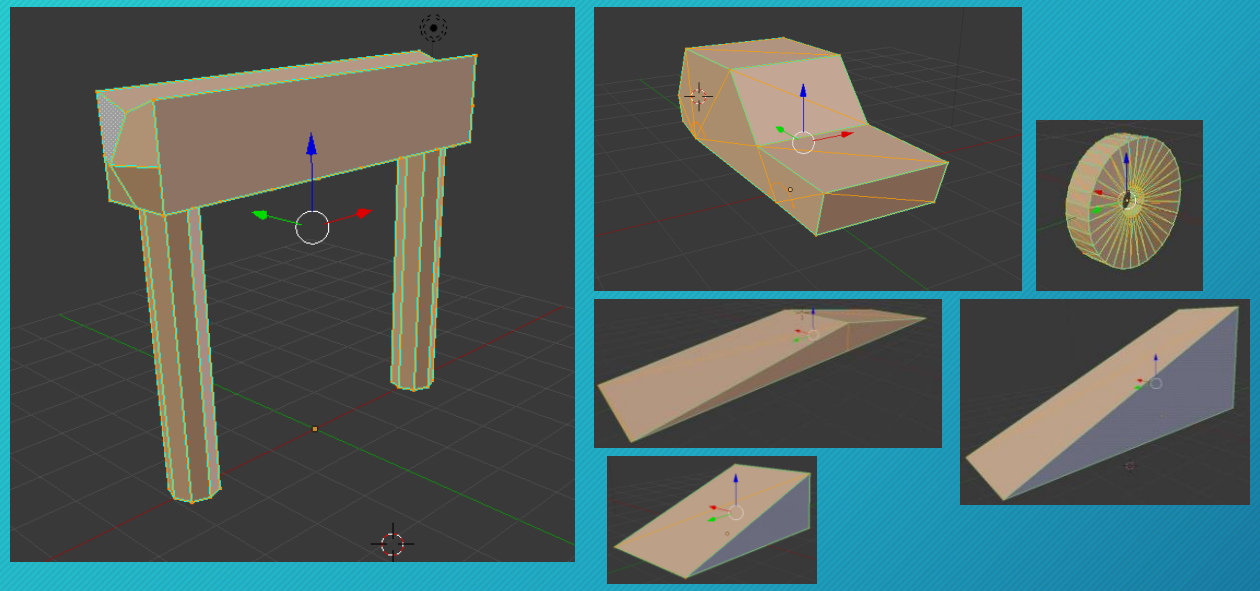
- If you land on your roof, you'll lose!
- But don't worry, if you're feeling stuck or you want to play again you can click reset whenever you want.

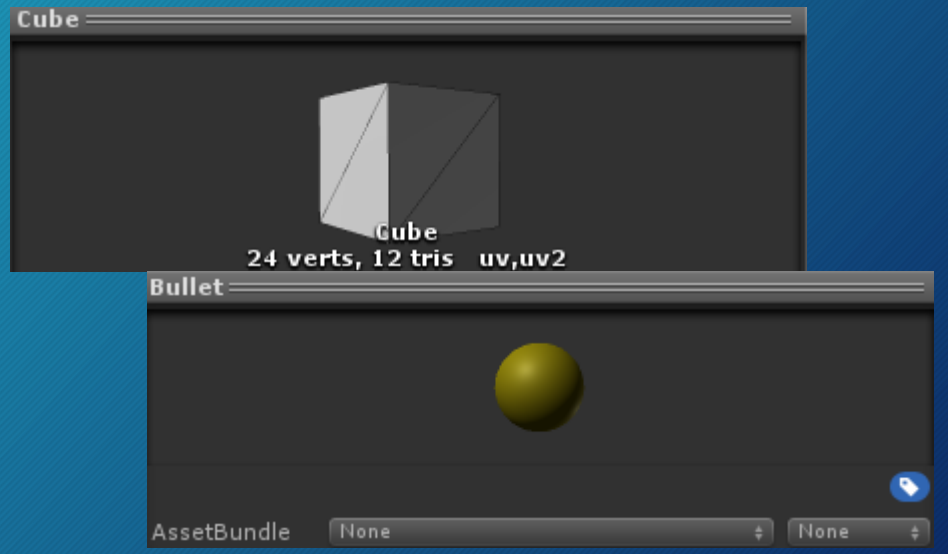
BASE

How we implemented each of the base requirements

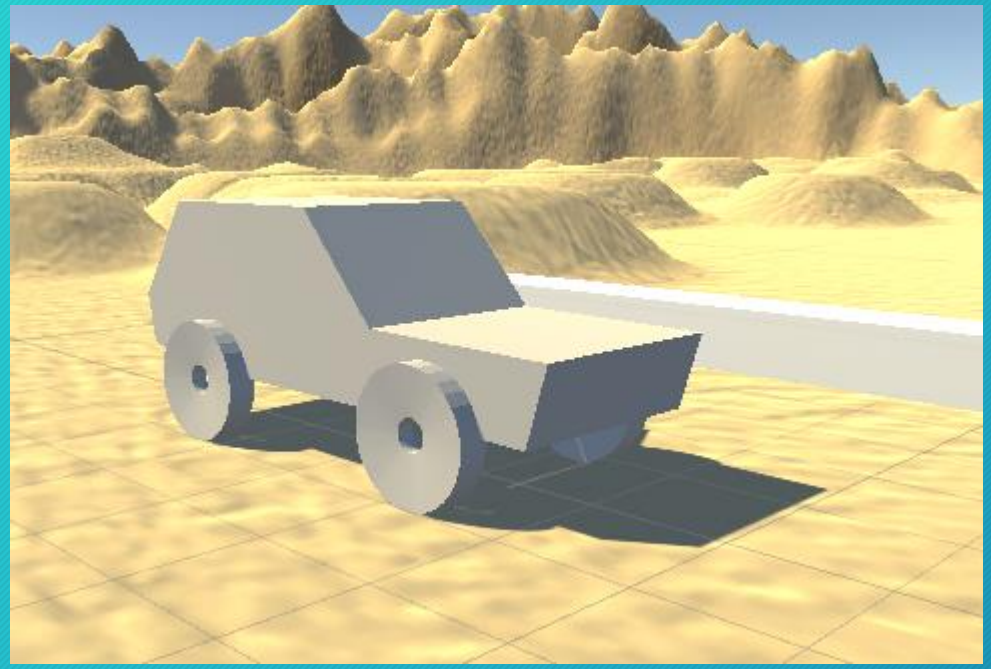
BASE

- We put together our game scene using just a few objects we made using Blender
- As well as basic meshes offered in Unity, like the cube and sphere



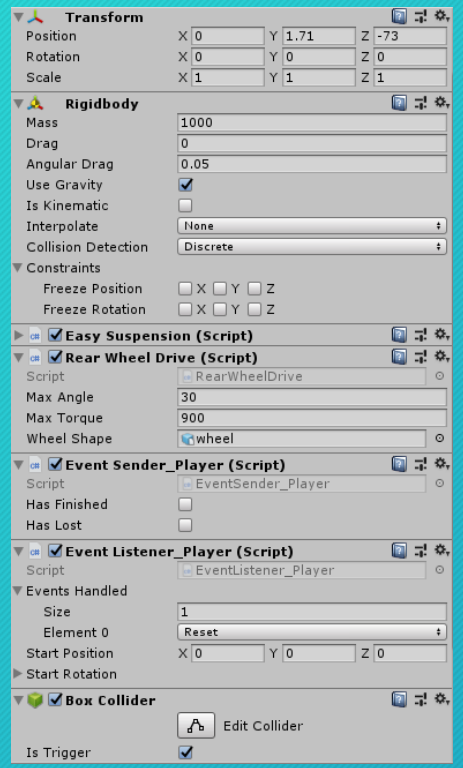


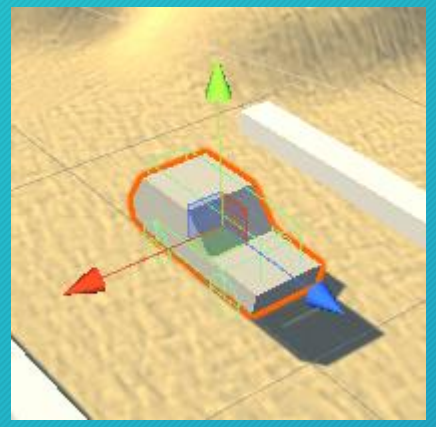
The Car



- We created a 1:1 scale accurate to life replica of the 1970 AMC Gremlin to serve as our moveable, controllable main character

The Car





- The Gremlin contains a RigidBody component which allows it to be affected by gravity
- We made the car and model wheel ourselves
- It's axel and wheelbase were made using scripts from the free to use Unity Vehicle Tools project, allowing us to use the BoxyCarWizard

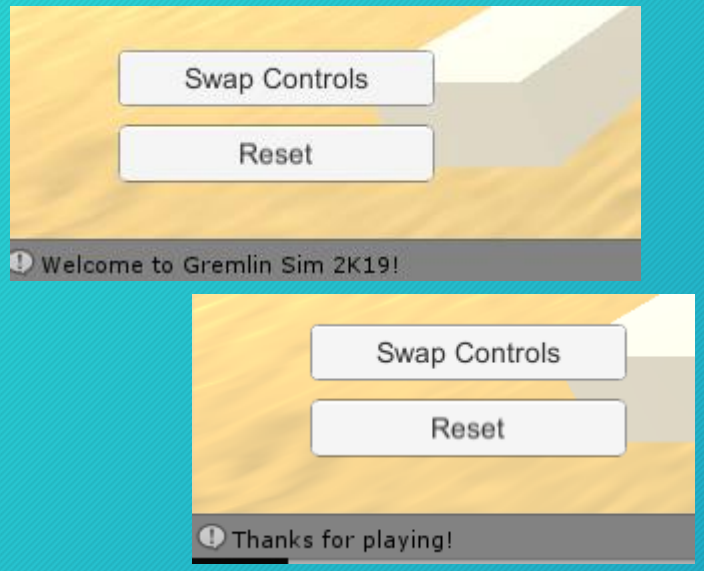
The Car

```
0 references
public void Update()
{
    //float angle = maxAngle * Input.GetAxis("Horizontal");
    //float torque = maxTorque * Input.GetAxis("Vertical");

    float angle = maxAngle * Command.X_Axis;
    float torque = maxTorque * Command.Y_Axis;

    foreach (WheelCollider wheel in wheels)
    {
        // a simple car where front wheels steer while rear ones drive
        if (wheel.transform.localPosition.z > 0)
            wheel.steerAngle = angle;

        if (wheel.transform.localPosition.z < 0)
            wheel.motorTorque = torque;
```

RearWheelDrive.cs

- The changes we made can be seen here
- We used our own axis modified by our commands to adjust the steering angle and torque instead of Unity's Input Axis
- This allowed us to customize the controls

Use of DLL



- We implemented the handy dandy Greeting DLL we learned to make in our tutorial
- Pressing ‘H’ and ‘G’ print messages into the console
- The messages can be set in Unity through a C# script

Use of DLL

```cpp
#include "Greeter.h"

char* Greeter::greet()
{
    return (char*)greeting->c_str();
}

void Greeter::setGreeting(std::string g)
{
    delete greeting;
    greeting = new std::string(g);
}

int Greeter::add(int first, int second)
{
    return first + second;
```

```cpp
#pragma once
#include "LibSettings.h"
#include "Greeter.h"

#ifdef __cplusplus
extern"C"
{
#endif

    LIB_API int Add(int first, int second);
    LIB_API char* SayHello();
    LIB_API void SetGreeting(char* greeting);

#ifdef __cplusplus
}
#endif
```

```cpp
#include "Wrapper.h"

Greeter greeter;

int Add(int first, int second)
{
    return greeter.add(first, second);
}

char* SayHello()
{
    return greeter.greet();
}

void SetGreeting(char* greeting)
{
    greeter.setGreeting(std::string(greeting));
}
```

- Basically it works by defining some functions in c++ using only things that can be translated into C
- We then write “Wrapper” functions, in C that return the functions we wrote in C++
- These can then be called upon in Unity

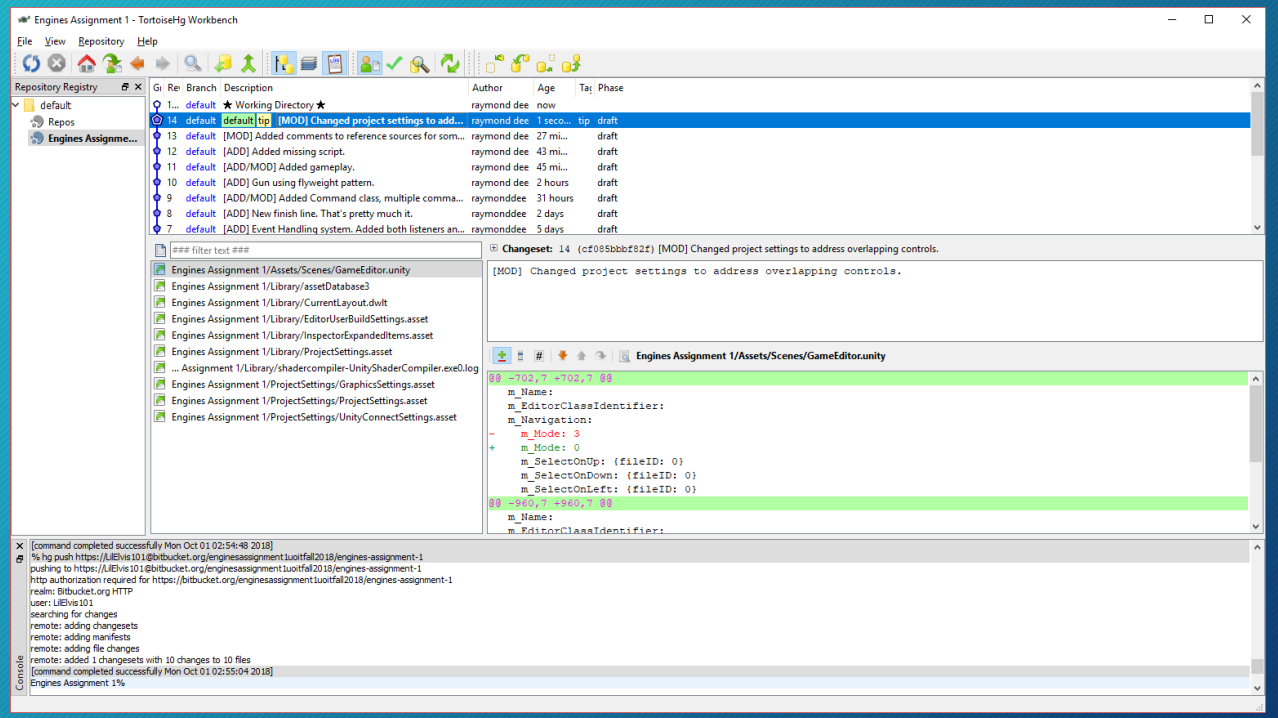
Use of DLL

```
public class HelloWorldPluginWrapper : MonoBehaviour
{
    const string DLL_NAME = "HelloWorldPlugin";

    // Use this for initialization

    [DllImport(DLL_NAME)]
    1 reference
    private static extern int Add(int first, int second);
    [DllImport(DLL_NAME)]
    2 references
    private static extern System.IntPtr SayHello();
    [DllImport(DLL_NAME)]
    2 references
    private static extern void SetGreeting(string greeting);

    // Update is called once per frame
    0 references
    void Update ()
    {
        if(Input.GetKeyDown(KeyCode.H))
        {
            print(Add(2000,19));

            SetGreeting("Welcome to Gremlin Sim 2K19!");

            string message = Marshal.PtrToStringAnsi(SayHello());
            print(message);
        }

        if (Input.GetKeyDown(KeyCode.G))
        {
            SetGreeting("Thanks for playing!");

            string message = Marshal.PtrToStringAnsi(SayHello());
```

- These functions can then be imported from the DLL we compiled
- We can print numbers without much effort, but strings need to be “Marshalled”

Version Control

- We used Tortise HG to manage our repository
- Pushing to it often every time progress was made in order to ensure if anything went wrong we could revert to a previous build
- We used an .hg ignore file to ensure the repo stayed clean.



Part 1: Command Design Pattern

Using the command pattern to change the controls

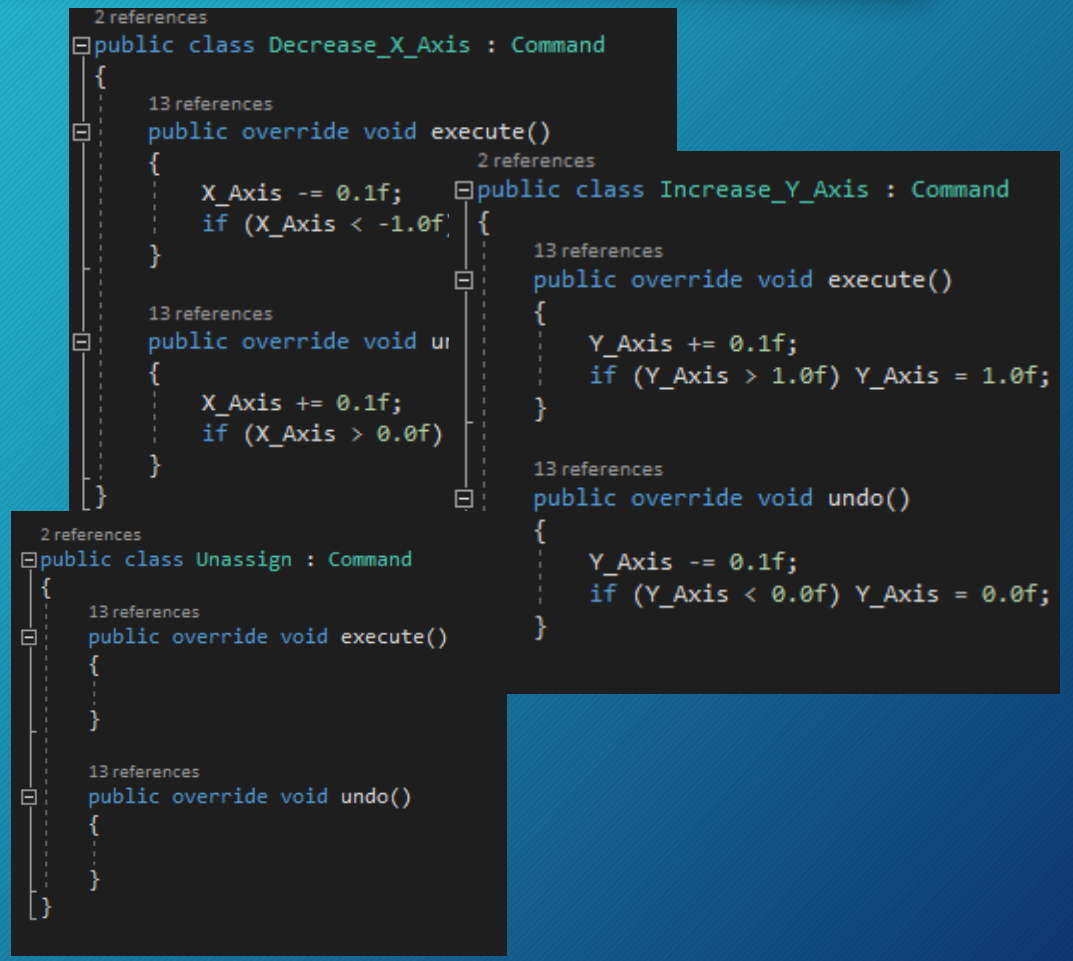
Command Class

```
15 references
public abstract class Command
{
    13 references
    public abstract void execute();
    13 references
    public abstract void undo();

    public static float X_Axis = 0.0f;
    public static float Y_Axis = 0.0f;
}
```

- Our implementation of the command pattern is very simple
- We have an abstract Command base class
- It contains abstract functions execute() and undo(); as well as two static floats used to store axis all the children interact with

Command Children

- There are five “commands” each represented by a child of Command
- They either increase, decrease, or unassign an axis to each command



Input Handler

```
public class InputHandler : MonoBehaviour
{
    static public Increase_X_Axis IncreaseX = new Increase_X_Axis();
    static public Increase_Y_Axis IncreaseY = new Increase_Y_Axis();
    static public Decrease_X_Axis DecreaseX = new Decrease_X_Axis();
    static public Decrease_Y_Axis DecreaseY = new Decrease_Y_Axis();
    static public Unassign Unassign_Axis = new Unassign();
    static public Command buttonW = IncreaseY;
    static public Command buttonS = DecreaseY;
    static public Command buttonA = DecreaseX;
    static public Command buttonD = IncreaseX;
    static public Command buttonUP = Unassign_Axis;
    static public Command buttonDOWN = Unassign_Axis;
    static public Command buttonLEFT = Unassign_Axis;
    static public Command buttonRIGHT = Unassign_Axis;
    1 reference
    public void handleInput()
    {
        if (Input.GetButton("W")) buttonW.execute();
        if (Input.GetButton("S")) buttonS.execute();
        if (Input.GetButton("A")) buttonA.execute();
        if (Input.GetButton("D")) buttonD.execute();
        if (Input.GetButton("UP")) buttonUP.execute();
        if (Input.GetButton("DOWN")) buttonDOWN.execute();
        if (Input.GetButton("LEFT")) buttonLEFT.execute();
        if (Input.GetButton("RIGHT")) buttonRIGHT.execute();

        if(!Input.anyKey)
        {
            buttonW.undo();
            buttonS.undo();
            buttonA.undo();
            buttonD.undo();
            buttonUP.undo();
            buttonDOWN.undo();
            buttonLEFT.undo();
            buttonRIGHT.undo();
        }
    }

    0 references
    public void Update()
    {
        handleInput();
    }
}
```

- We created a custom InputHandler class that contains objects of type Command for each key, and one of each child type
- Since classes are reference types in C# by default, we can simply reassign each of the buttons to whatever child of command we want

Swapping Controls

- When the button to swap controls on the UI is clicked, the controls are assigned or unassigned appropriately
- This logic is done in the InputHandler's event listener

```
if (messageType == EventRelay.EventMessageType.SwapControls)
{
    wasd = !wasd;

    if (wasd)
    {
        InputHandler.buttonW = InputHandler.IncreaseY;
        InputHandler.buttonS = InputHandler.DecreaseY;
        InputHandler.buttonA = InputHandler.DecreaseX;
        InputHandler.buttonD = InputHandler.IncreaseX;
        InputHandler.buttonUP = InputHandler.Unassign_Axis;
        InputHandler.buttonDOWN = InputHandler.Unassign_Axis;
        InputHandler.buttonLEFT = InputHandler.Unassign_Axis;
        InputHandler.buttonRIGHT = InputHandler.Unassign_Axis
    }
    else
    {
        InputHandler.buttonW = InputHandler.Unassign_Axis;
        InputHandler.buttonS = InputHandler.Unassign_Axis;
        InputHandler.buttonA = InputHandler.Unassign_Axis;
        InputHandler.buttonD = InputHandler.Unassign_Axis;
        InputHandler.buttonUP = InputHandler.IncreaseY;
        InputHandler.buttonDOWN = InputHandler.DecreaseY;
        InputHandler.buttonLEFT = InputHandler.DecreaseX;
        InputHandler.buttonRIGHT = InputHandler.IncreaseX;
    }
}
```

Part 2: Flyweight Design Pattern

How we implemented the flyweight pattern to make masses of bullets

Sharing Meshes

- Basically, within our Bullet script, we make the object's mesh equal to the sharedMesh of the Mesh Filter component
- This ensures that all the objects are reading the data of a single mesh
- We verified this by accidentally modifying the mesh data at runtime, and seeing it persist between play sessions, meaning that each bullet really was using the same mesh (We have since corrected this)

```
public class Bullet : MonoBehaviour
{
    public float scaleFactor = 2.0f;

    0 references
    void Start ()
    {
        Mesh mesh = GetComponent<MeshFilter>().sharedMesh;
        //Mesh mesh = new Mesh(); //THE BAD WAY
        Vector3[] vertices = mesh.vertices;
        int p = 0;
        while(p < vertices.Length)
        {
            vertices[p] *= scaleFactor;
            p++;
        }
        mesh.vertices = vertices;
        mesh.RecalculateNormals();
    }
```

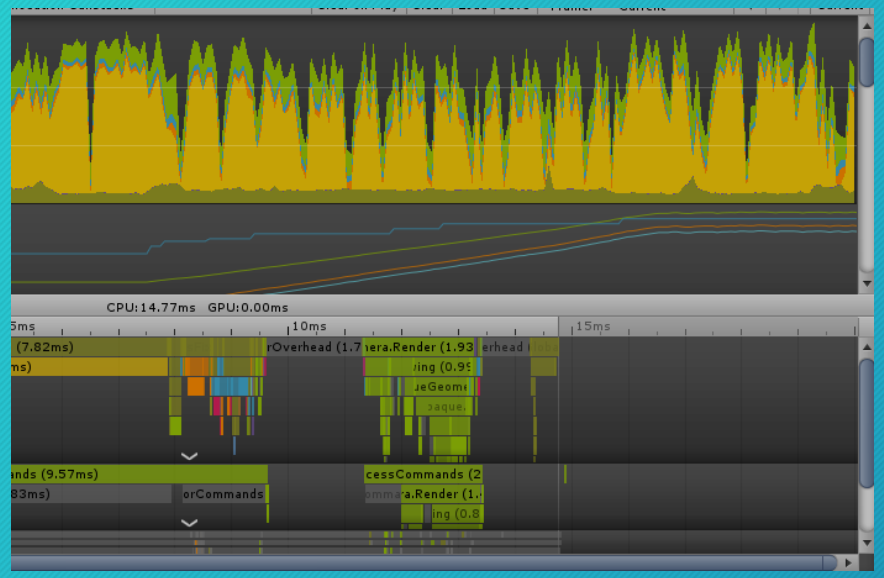
Instantiating

- We attached the Bullet script to a bullet prefab object
- We then created a Gun class (a factory) that essentially just instantiates these prefabs with a velocity

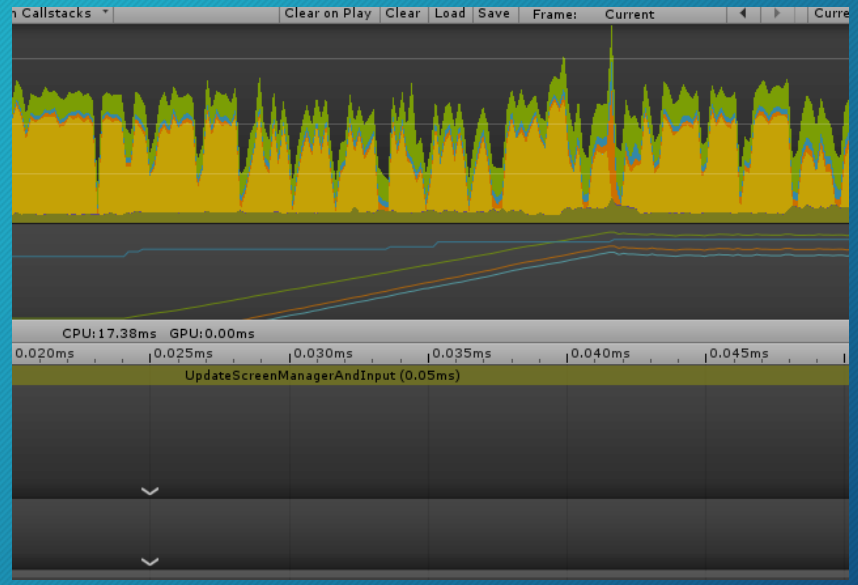
```
void Fire()
{
    var bullet = (GameObject)Instantiate(bulletPrefab, bulletSpawn.position, bulletSpawn.rotation);

    bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward * 12;

    Destroy(bullet, 2.0f);
}
```

Profiler Comparison

Mesh mesh = new Mesh();



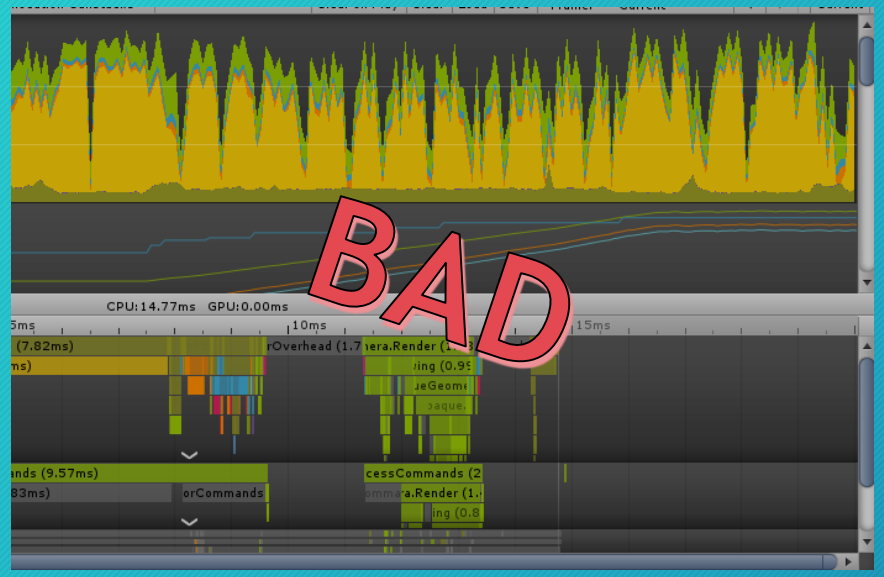
Mesh mesh = GetComponent<MeshFilter>().sharedMesh;



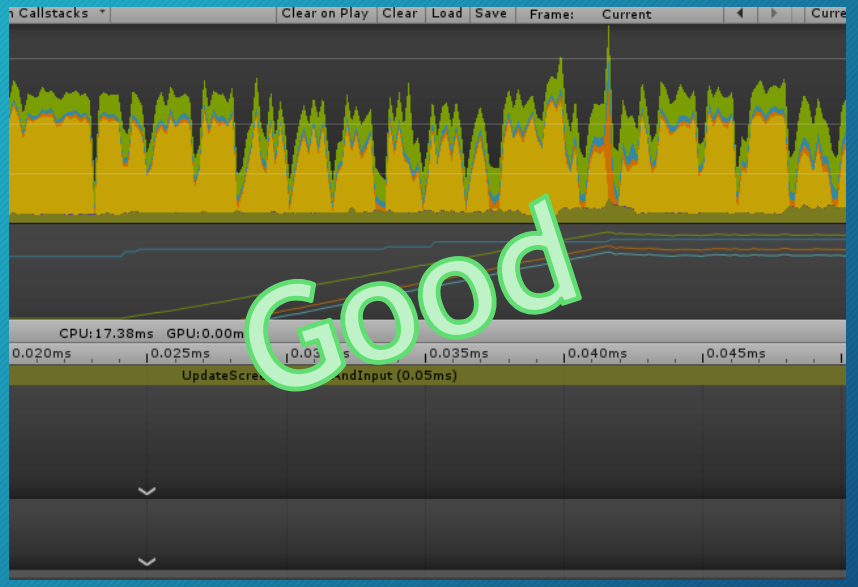
Here's a comparison of Unity's profiler with flyweight on the right, and without on the left

Profiler Comparison

Mesh mesh = new Mesh();



Mesh mesh = GetComponent<MeshFilter>().sharedMesh;



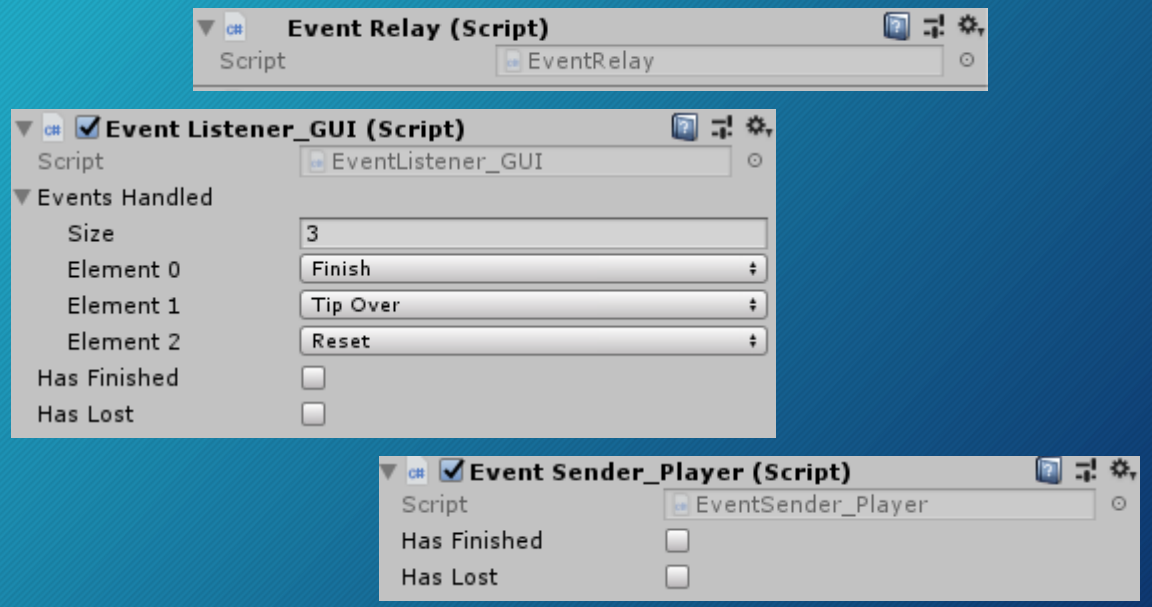
The morale of the story is that making new meshes is much more taxing

Part 3: Observer Pattern

How we used the observer pattern to do EVERYTHING else

Event System

- Our event system is composed of several classes, it's based on a tutorial we followed on LinkedInLearning
- There is a single EventRelay class that relays events between any sender and any listener
- There can be as many EventSender and EventListener classes as there are objects that need to send or listen for events
- Not every object needs one, and not every object that has a listener needs a sender, or visa versa



Event Relay

- The EventRelay class contains four things: a delegate, an event, an enum of event types, and a function RelayEvent
- RelayEvent() simply takes in an event type and the name of the sender and passes it to the delegate
- The delegate contains all of the listener's handle events functions which have the exact same parameters

```
33 references
public class EventRelay : MonoBehaviour
{
    public delegate string EventAction(EventMessageType type, MonoBehaviour sender);
    public static event EventAction OnEventAction;

    23 references
    public enum EventMessageType
    {
        Finish,
        TipOver,
        Reset,
        SwapControls
    }

    4 references
    public static string RelayEvent(EventMessageType messageType, MonoBehaviour sender)
    {
        return OnEventAction(messageType, sender);
    }
}
```

Event Senders

- The sender classes contain a variety of things, ranging from variables to store statuses of events and any logic needed to fire an event
- Usually an update and any sort of trigger functions

```
public class EventSender_GUI : MonoBehaviour
{
    public bool resetClicked = false;
    public bool swapClicked = false;

    0 references
    void Update()
    {
        if(resetClicked)
        {
            string value = EventRelay.RelayEvent(EventRelay.EventMessageType.Reset, this);
            Debug.Log("Reset Event was seen by: " + value);
            resetClicked = false;
        }

        if (swapClicked)
        {
            string value = EventRelay.RelayEvent(EventRelay.EventMessageType.SwapControls, this);
            Debug.Log("SwapControls Event was seen by: " + value);
            swapClicked = false;
        }
    }

    0 references
    public void OnResetButtonClick()
    {
        resetClicked = true;
    }

    0 references
    public void OnSwapButtonClick()
    {
        swapClicked = true;
    }
```

EventSender_Player

- The player's event sender contains an update function that sends the event that it has won or lost depending on if a couple of bools
- It also contains an OnTrigger function that modifies these bools depending on what it's colliding with (the finish line with the car, or the roof with the ground)

```
0 references
public class EventSender_Player : MonoBehaviour
{
    public bool hasFinished = false;
    public bool hasLost = false;

    0 references
    void Update()
    {
        if(hasFinished)
        {
            string value = EventRelay.RelayEvent(EventRelay.EventMessageType.Finish, this);
            Debug.Log("Finished Event was seen by: " + value);
            hasFinished = false;
        }
        if(hasLost)
        {
            string value = EventRelay.RelayEvent(EventRelay.EventMessageType.TipOver, this);
            Debug.Log("TipOver Event was seen by: " + value);
            hasLost = false;
        }
    }

    0 references
    public void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Finish")
            hasFinished = true;

        if (other.tag == "Ground")
            hasLost = true;
    }
}
```

Event Listeners

- The listener classes contain at least five things: a list of all the events they are subscribed to, an OnEnable and OnDisable function, a HandleEvent function, and a function that prints it's status to the debug log

```
public class EventListener_GUI : MonoBehaviour
{
    public List<EventRelay.EventMessageType> eventsHandled = new List<EventRelay.EventMessageType>();

    public bool hasFinished = false;
    public bool hasLost = false;

    GUIStyle styleVictory = new GUIStyle();
    GUIStyle styleDefeat = new GUIStyle();

    0 references
    void OnEnable()
    {
        EventRelay.OnEventAction += HandleEvent;
    }

    0 references
    void OnDisable()
    {
        EventRelay.OnEventAction -= HandleEvent;
    }

    0 references
    void Start()
    {
        styleVictory.alignment = TextAnchor.MiddleCenter;
        styleVictory.fontSize = 68;
        styleVictory.normal.textColor = new Color(0.5f, 1.0f, 0.5f, 1.0f);

        styleDefeat.alignment = TextAnchor.MiddleCenter;
        styleDefeat.fontSize = 68;
        styleDefeat.normal.textColor = new Color(1.0f, 0.5f, 0.5f, 1.0f);
    }

    0 references
```

Event Listeners

- The OnEnable function and OnDisable function add and remove the HandleEvent function to and from the EventRelay's delegate respectively

```
void OnEnable()
{
    EventRelay.OnEventAction += HandleEvent;
}

0 references
void OnDisable()
{
    EventRelay.OnEventAction -= HandleEvent;
}
```

Event Listeners

- The HanldeEvent function implements any logic that should be done whenever an event in the subscription list is heard

```
string HandleEvent(EventRelay.EventMessageType messageType, MonoBehaviour sender)
{
    if(messageType == EventRelay.EventMessageType.Finish)
    {
        hasFinished = true;
    }

    if(messageType == EventRelay.EventMessageType.TipOver)
    {
        hasLost = true;
    }

    if (messageType == EventRelay.EventMessageType.Reset)
    {
        hasFinished = false;
        hasLost = false;
    }

    if (eventsHandled.Contains(messageType))
    {
        Debug.Log("Handled Event: " + messageType + " from sender: " + sender);
        return this.ToString();
    }
    else
    {
        return this.ToString();
    }
}
```

EventListener_GUI


```
public class EventListener_GUI : MonoBehaviour
{
    public List<EventRelay.EventMessageType> eventsHandled = new List<EventRelay.EventMessageType>();

    public bool hasFinished = false;
    public bool hasLost = false;

    GUIStyle styleVictory = new GUIStyle();
    GUIStyle styleDefeat = new GUIStyle();

    0 references
    void OnEnable()
    {
        EventRelay.OnEventAction += HandleEvent;
    }

    0 references
    void OnDisable()
    {
        EventRelay.OnEventAction -= HandleEvent;
    }
```

```
    0 references
    void OnGUI()
    {
        if(hasFinished)
        GUI.Label(new Rect(200, 200, 300, 50), "VICTORY", styleVictory);

        if(hasLost)
        GUI.Label(new Rect(200, 200, 300, 50), "DEFEAT", styleDefeat);
    }

    2 references
    string HandleEvent(EventRelay.EventMessageType messageType, MonoBehaviour sender)
    {
        if(messageType == EventRelay.EventMessageType.Finish)
        {
            hasFinished = true;
        }

        if(messageType == EventRelay.EventMessageType.TipOver)
        {
            hasLost = true;
        }

        if (messageType == EventRelay.EventMessageType.Reset)
        {
            hasFinished = false;
            hasLost = false;
        }

        if (eventsHandled.Contains(messageType))
        {
            Debug.Log("Handled Event: " + messageType + " from sender: " + sender);
            return this.ToString();
        }
        else
```

- The GUI's event listener class listens for the Finish, Tip Over, and Reset events
- It then flags bools when the events are heard
- And draws different things on the screen depending on which bools are flagged

Part 4: Pseudocode/UML

*Plans for the command pattern is included in the same file as this report