Input Systems (C#)

SUSHI ROLL MOBILE GAME – ALEX MEESTERS (142783)

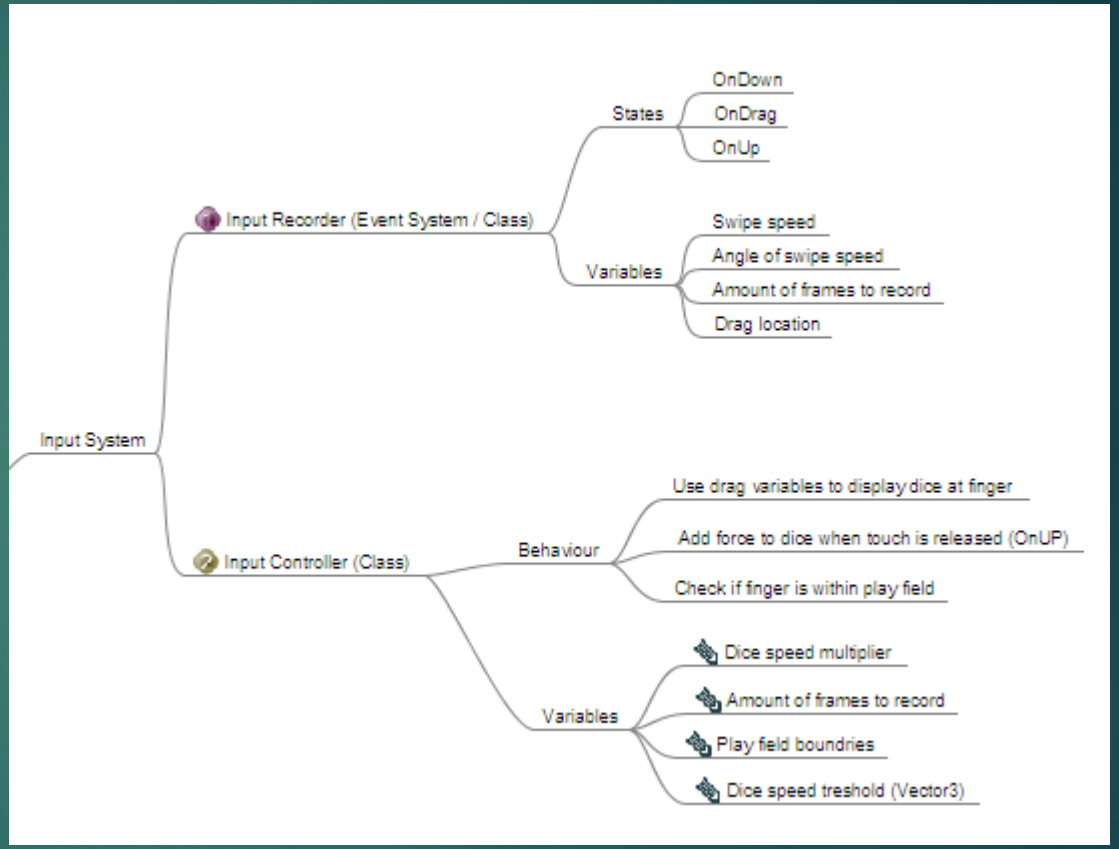
Input system

The input system is the most important system for the core gameplay of the game. Because the game is a physics based game, where direction and velocity play a very big role.

I have separated this system into two classes:

- Input Recorder (input)
- Input Controller (output)

The reason for separating the classes is to maintain overview of the project.



Created with FreeMind software

How to tackle input with mobile devices

There are two important factors for input
With mobile devices:

- Screen size
- Screen resolution

Should I calculate speed on X and Y locations?
This is a bad idea for an game that needs to take swipe velocity in mind.

This is because of the relation of screen size and screen resolution.
Having a smaller screen means a higher pixel density. Which means more distance can be travelled with a shorter finger movement.

What are the possible solutions for this problem? (Analysis on next page)

- Ray cast to a plane in game space
- Normalize pixel values by dividing them with the screen resolution.

Solutions for tracking input

Ray cast to a plane in game space
This does almost the same as returning a normalized pixel value. (With a orthographic camera)
The only difference is that the ray cast data can be put to use.
By for example using the input data to show an object.
Ray casts are expensive calculations, but are accepted if used to a minimum.

Normalized pixel values
Touching the middle of screen would return (X0.5,Y0.5) for all devices. This eliminates the pixel density problem.
Although this would still mean that for bigger devices a longer swiping distance is required to get the same results.

Take DPI into account
Unity has a Screen.DPI function. Although this has not been proven to always work correctly. So it should rely on a failsafe system if it returns 0.

My choice

I have chosen to use the event system of Unity 3d for input. This system works based on ray casts.

Why?
Using ray casts is more expensive. But they immediately do provide intersection information. Which I can use to display the dice at the ray cast point, without needing to convert input information to 3d space.

InputReader.cs

The data is obtained with the interfaces from the

UniteEngine.Eventsystems API:
* IPointerDownHandler
* IDragHandler
* IPointerUpHandler

This was set up by adding an physics ray caster to the main camera with a mask. And creating a GameOBJ with a box collider with the corresponding layer for the mask.

The input recorder will then send the input data that is obtained from the event system to the Input Controller class.

```
void Start ()
{
    GameCamera = Camera.main;
    InputCtrl = ClassController.InputCtrl;
    FrameCount = this.gameObject.AddComponent<FrameCounter> ();
    FrameCount.SetVectorCap (10); // Sets the amount of frames it needs to record.
}

void ResetValues ()
{
    prevPointWorldSpace = Vector3.zero;
    currentPointWorldSpace = Vector3.zero;
    realWorldTravel = Vector3.zero;
    FrameCount.ClearList ();
}

public void OnPointerDown (PointerEventData data)
{
    ResetValues ();
    prevPointWorldSpace = GameCamera.ScreenToWorldPoint (data.position);
    InputCtrl.MoveDice ("DOWN", data.pointerCurrentRaycast.worldPosition);
}

public void OnDrag (PointerEventData data)
{
    InputCtrl.MoveDice ("DRAG", data.pointerCurrentRaycast.worldPosition);

    currentPointWorldSpace = GameCamera.ScreenToWorldPoint (data.position);
    realWorldTravel = currentPointWorldSpace - prevPointWorldSpace;
    realWorldTravel.y = 0;

    prevPointWorldSpace = currentPointWorldSpace;

    FrameCount.Addvector (realWorldTravel);
}

public void OnPointerUp (PointerEventData data)
{
    InputCtrl.MoveDice ("UP", (FrameCount.GetDirection () * FrameCount.GetAverageVelocity ()));
}
```

This is bad, you do not want to compare strings. Using an enumeration would be better.

FrameCounter.cs

This is written to obtain information of X amount of swiping data.
The reason for the creation of this script was because the input was very inaccurate. It used only one frame of data, which caused the dice to move very randomly.

I have decided to store the Vector3 positions within a List instead of an Array. This is because it is easier to maintain.
Because:
* The amount of elements are dynamic.
* I can easily see how many elements are from a new recording

```
public void Addvector (Vector3 Addvector)
{
    CheckListSize ();
    if (Vector3.Dot (Camera.main.transform.position, Addvector) < 0) {
        VectorPositions.Add (Addvector);

    } else {
        if (VectorPositions.Count > 0) {
            ClearList ();
        }
    }
}

public float GetAverageVelocity ()
{
    AverageVelocityVector = Vector3.zero;

    GetListLength = VectorPositions.Count;

    if (GetListLength > 0) {
        for (int i = 0; i < GetListLength; i++) {
            AverageVelocityVector += VectorPositions [i];
        }

        AverageVelocityVector /= GetListLength;
        AverageVelocityFloat = AverageVelocityVector.z;

        return Mathf.Abs (AverageVelocityFloat) + 1;
    } else
        return 0;
}

public Vector3 GetDirection ()
{
    if (VectorPositions.Count > 0) {
        FirstFrame = VectorPositions [0];
        LastFrame = VectorPositions [VectorPositions.Count - 1];
        Direction = LastFrame - FirstFrame;

        Direction.y = 0;

        print (VectorPositions.Count - 1);

        return Direction.normalized;
    } else {
        return Vector3.zero;
    }
}
```

Used dot product to see if the velocity was not facing towards the camera (Camera rotates in game)

Used basic math to calculate swipe speed. (Could still be Improved by taking normalized Screen movement into account As well, since only frames within The input field are recorded)

Used basic math to calculate vector direction

FrameCounter.cs

Changes to the code based on feedback:

The stored vectors are directional vectors. I renamed VectorPositions to DirectionalVectors.

I changed the function of GetAverageVelocity to have a string input of an “Axis”. So that it returns the corresponding information based on the string.

I also removed the +1 that got added to the AverageVelocityFloat

```
public void Addvector (Vector3 Addvector)
{
    CheckListSize ();
    if (Vector3.Dot (Camera.main.transform.position, Addvector) < 0) {
        DirectionalVectors.Add (Addvector);

    } else {
        if (DirectionalVectors.Count > 0) {
            ClearList ();
        }
    }
}

public float GetAverageVelocity (string axis)
{
    GetListLength = DirectionalVectors.Count;

    if (GetListLength > 0) {

        AverageVelocityVector = Vector3.zero;

        for (int i = 0; i < GetListLength; i++) {
            AverageVelocityVector += DirectionalVectors[i];
        }

        AverageVelocityVector /= GetListLength;

        switch (axis) {
            case "z": AverageVelocityFloat = AverageVelocityVector.z; break;
            case "x": AverageVelocityFloat = AverageVelocityVector.x; break;
        }

        return Mathf.Abs (AverageVelocityFloat);
    } else
        return 0;
}

public Vector3 GetDirection ()
{
    if (DirectionalVectors.Count > 0) {
        FirstFrame = DirectionalVectors[0];
        LastFrame = DirectionalVectors[DirectionalVectors.Count - 1];
        Direction = LastFrame - FirstFrame;

        Direction.y = 0;

        return Direction.normalized;
    } else {
        return Vector3.zero;
    }
}
```

Problems with current input system

Method of obtaining direction is bad
When swiping the dice over the mobile screen: the current system will use the first added directional vector and the last. And calculate the direction from the first to the last.

The reason why it is bad to use directional vectors to create a direction vector, is because they do not represent the actual dice begin and end position. They already represent movement changes from frame to frame. This also created inaccuracies within Unity.

Method of obtaining velocity is bad
The current function adds all the directional vectors and returns the average length of the vector. And then just takes the length of the Z position (based on input string). It did this because I wanted to prevent motion along the X axis.

This problem was dealt with the wrong way. Because it would mean that the finger motion would not always represent the actual motion. For example when you want to throw a dice in an corner.

Solution for velocity

Instead of only getting a certain axis of movement, I use the dot product to only allow throwing of dice for a certain angle towards the playing field.

Doing this makes the movement more accurate because angles get taken into account.

The new way of getting the average velocity of the positional vectors is by calculating the distance between the two vectors, for each vector in the list. And then dividing that with the total amount of positional vectors.

My previous method had already taken the velocity into account. When adding them to the list.

```
public float GetAverageVelocity ()
{
    AverageVelocityVector = Vector3.zero;

    GetListLength = VectorPositions.Count;

    if (GetListLength > 0) {
        for (int i = 0; i < GetListLength; i++) {
            AverageVelocityVector += VectorPositions [i];
        }

        AverageVelocityVector /= GetListLength;
        AverageVelocityFloat = AverageVelocityVector.z;

        return Mathf.Abs (AverageVelocityFloat) + 1;
    } else
        return 0;
}
```

```
bool IsFacingAwayFromCam (Vector3 directionalVector)
{
    float MinimumCameraAngle = Vector3.Dot (Camera.main.transform.position, directionalVector);

    if (MinimumCameraAngle < -3.75f) {

        return true;
    } else
        return false;
}
```

```
public float GetAverageVelocity ()
{
    GetListLength = PositionalVectors.Count;

    if (GetListLength > 0) {

        float AverageSpeed = 0;

        for (int i = 0; i < GetListLength; i++) {
            if (i+1 <= GetListLength -1) {
                AverageSpeed += Vector3.Distance(PositionalVectors[i+1], PositionalVectors[i]);
            }
        }

        return AverageSpeed/GetListLength;

    } else return 0;
}
```

Solution for direction

What I did before was using this method for directional vectors. Which is bad because a directional vector is already the end result of the calculation of two positional vectors.

This is more accurate because the positional vectors represent the begin point and the end point of movement. From which I create a directional vector.

```
public Vector3 GetDirection ()
{
    if (PositionalVectors.Count > 0) {
        FirstFrame = PositionalVectors [0];
        LastFrame = PositionalVectors [PositionalVectors.Count - 1];
        Direction = LastFrame - FirstFrame;

        Direction.y = 0;

        return Direction.normalized;
    } else {
        return Vector3.zero;
    }
}
```

Card system

The game also requires a card dragging system. What this means is that the card input is the first step, before triggering the dice input based systems.

When grabbing the card it also meant that the eyes of the dice would change according to the popup information. It would also add an active multiplier to the scene.

Receiving the input for the card system.

For the cards I used the same interface as the InputReader.cs. The input field will get activated after dragging the card for a certain distance. After this is done, the pointer data will be send to the InputReader.cs.

After dragging the card:

- I call the sound manager to play a pickup sound
- I disable the picked card using the UI card manager
- Activate the input field
- I fade all cards except for the used index with the UI card manager.

```
public void OnDrag (PointerEventData data)
{
    if (Input.touchCount == 1 || Application.isEditor) {

        if (!hasDraggedCard && Vector3.Distance (DevideByScreenResolution (data.position), DevideByScreenResolution (pointerStartPosition)) > 0.015f) {

            soundManager.PlaySoundEffect (soundManager.soundDatabase.uiCardSelect, 1, 1);

            uiCardManager.ShowInfoDisplay (false, Vector3.zero);

            thisCardDisplayer.EnableCardImages (false, HasUsedCard [sessionManager.GetCurrentSession ().activePlayer - 1]);
            inputField.Activate (true);
            hasDraggedCard = true;
            uiCardManager.FadeCardsExeptIndex (CardOBJIndex, true);

        } else
            inputReader.OnDrag (data);
    }

}
```

What have I learned?

Talking about your problems with others can help you solve the problem better and quicker
During the development of the input I was stuck on how to improve it.
It felt clunky and it sometimes seemed to move towards the wrong directions. This was mainly because only one frame got recorded.
The moment I started talking to another programmer about the problem. I basically gave him the answer.
(Rubber duck problem solving)

Why lists can be more useful than arrays
The dynamic nature of lists can be a very useful tool, not having to manually reset values. Or needing other variables to keep track of the latest place.

Refreshed my memory on vectors
What I have done is not very maths heavy, since Unity provides a lot of functionality. But I was forced to keep a mental image, knowing how I would make the solutions work. Knowing the differences between velocity, world space, local space. And how it would apply towards a physics engine in Unity 3D

What have I learned?

Many complications can come up within the development process

Developing the input system was not a very easy thing to do. After completing the basic system I ran into complications. This was because I had to implement card dragging. Which was also based on UI ray casting. This required me to find new solutions based on a system that I had previously made.