Vertical Slice

Week 1 (5.11. >)

This is the first week of the Vertical Slice that I think is also like a transition week because we officially haven't in class started the vertical slice. But in this week there is the Playtesting Evening.

This week we had a guest lecture from a guy called Luca about his roguelike game he is developing and he also gave us feedback on our game. I would boil down the feedback to saying that our game just 'exists', it is just another roguelike and there is nothing special to it. After he brought that forward, I realized that it is so. Our game is just a combination of different games. We are lacking a 'thing'.

Playtesting Evening

There are 3 main questions we had for the players trying our game.
Question 1: How do you like the shooting? Do you like auto aim or do you like manual shooting?
Question 2: What do you think about the health system, do you like the health points system and what do you think about the hit stop.
Question 3: What do you think about the camera? Is the player too small? Is the camera too far?

The people generally liked the auto aim more because they like to focus on the on the dodging more and it made the dodging more fun. My idea about having the auto aim was similar to that. However, at one point Eyal pointed out that if there is auto aim and there is no ammo mechanic when is there ever a reason to not hold down the button and not shoot. So even if the consensus is that auto aim might be better it gave us a lot to think about. That if we want shooting we might have to think about some resource management but that is another unnecessary difficulty on top.

The health system was well received, even though I didn't make the healing in time. One thing the people really didn't like was the hit stop. Although they understand it is to show that they got hit, they just didn't like they lost control for so long, partly because there was no animation so it felt like the game just froze.

And the camera, we already struggled with the size and scale. However Florian gave us some advice. He didn't like how our rooms are big and that the player is pretty small because of the camera. He gave us advice that we should start with smaller rooms and see how our camera fits with the amount of enemies and how much we see. But generally people thought that the player is pretty small and some had issues with the hitboxes,

however, those were conflicting. Some thought that the player is too big so dodging was hard but some people thought that the player is small so the dodging was easier.

One thing that gave me a piece of mind or even made me really happy was that Luca, the guest lecturer, told us that our team is probably the most solid of the teams that had the presentations. Even though it was a small comment, it made me happy and validated some of the work I have so far done.

Because our game just felt generic we already started thinking about what to do about it. I don't know who suggested but a potential idea was not having the shooting at all perhaps focusing on the movement/dashing more. That was certainly an interesting idea, because it would solve a lot of design problems we are having right now. Like, the game being generic shooter we would focus on the bullet hell part of the game and make it a little unique. It would solve the problem of choosing a shooting mode. And it could create some interesting combat. However, we were too tired to decide on anything and too soon to decide.

The Big Change Meeting

We had a meeting a few days after the Playtesting Evening about the feedback and how we should continue. The main problem was that our game just felt too generic. We also had a problem with the shooting. After a lengthy debate we have decided to throw out the shooting completely. We decided like this, so that we completely focus on the bullet hell aspect and make the game more about avoiding projectiles. Instead we would focus our combat system around the dashing ability. This would fix two of our problems, the problem with deciding on a shooting mode and the problem of having our game be generic. We then tried to brainstorm how to use the dash as a combat mechanic, not only just as a movement mechanic. Some proposed having the combat encounters just be surviving until a certain amount of time passes. However I countered, because having this just be the whole gameplay loop would be incredibly tedious and boring after a while. I think I proposed damaging enemies by dashing through them, however Sophie said this would just be replacing a ranged attack with something similar to a melee attack.
In the end Sophie brought forward a pretty nice idea. When the player dashes through an enemy it will mark them/apply a mark and after a while it will damage the enemy or do something. This kind of thing is pretty nice because we can make the abilities for this and make some pretty nice synergies with it. For example when the player dashes through an enemy and marks them it will apply a status effect that gives overtime damage, that also might apply a a slowness effect. Or an ability that creates an explosion once the mark ends combined with an ability that makes the mark end once the player dashes through the enemy while the enemy is marked.

Another very big change we decided on is to completely throw out the procedural dungeon aspect of the game. While it is something that would be nice to have we have a couple of reasons. First of all, I would have to redo the script so that it would allow for easier modification not just from designers but also for further enhancing of the script. I wouldn't have to then spend so much time on this and could focus on something else that is also

important, like the combat system. Other reason, it would allow us to do more stuff thematically. Of course we don't have to completely throw it out, we could make multiple versions of the map and just randomly choose which one the player will spawn in. This is a big change that alters our game in a major way.

Week 2 (12.11. >)

This week I did a minimal amount of work because there were a lot of events happening like Talk & Play and Games Ground, so I attended and volunteered on both of those. After that I was completely tired to do something. Nonetheless, I still finished the healing system so the player can now heal himself, we still haven't decided what will fill up the gauge now that the shooting is gone, so for now it's unlimited healing.

Week 3 (19.11. >)

I added the healing mechanic. Admittedly it is very similar to Hollow Knight's healing system but I believe this is something that fits for this game. It is risk based healing, you have to hold a button and wait a while to heal by one heart. It makes the player think when to heal during combat and reinforces the bullet hell aspect.

I also transitioned to the New Input System and learned a lot how to set up the code for that.

The main thing I created this week was the Mark system. My aim while doing this was to make it as flexible as possible so I can easily add new abilities, and make it work with Scriptable Objects so that that the abilities can easily be changed. It will function similarly to the Hades boon system. This week I am working only on the mark itself and the marked dash system where it executes an ability when dashing through an already marked enemy. There are 2 very important scripts excluding the abstract scripts of the abilities. The PlayerUpgradesManager and the EnemyMarkHandler.

```
public class PlayerUpgradesManager : MonoBehaviour
{
    public MarkBehavior equippedMark;  Default Mark Behavior.asset
    public OnDash equippedOnDash;  Unchanged
    public MarkedDash equippedMarkedDash;  Changed in 0+ assets
    public Passive equippedPassive;  Unchanged
    public float markDuration;  "8"

    2 usages
    public void ApplyMarkToEnemy(GameObject enemy)
    {
        EnemyMarkHandler markHandler = enemy.GetComponent<EnemyMarkHandler>();
        if (markHandler != null)
        {
            markHandler.markedDashAbility = equippedMarkedDash;

            markHandler.ApplyMark(equippedMark, gameObject, markDuration);
        }
        else
        {
            Debug.LogWarning("The enemy you are trying to dash doesn't have the EnemyMarkHandler script");
        }

    }
```

All the Manager does is store the abilities and apply the Marks to the enemies. I wanted one script to be the center for the stored abilities for ease of use.

```
Event function  usages  overrides  Simon SkvaraPC +1  extension methods
void Update()
{
    if (currentMark != null)
    {
        Elapsed += Time.deltaTime;

        currentMark.OnUpdate(gameObject, Time.deltaTime);

        RefillSoul();

        if (Elapsed > markDuration)
        {
            currentMark.OnExpire(enemy:gameObject, player);
            ClearMark();
            return;
        }

    }
}
```

```
1 usage  Simon SkvaraPC
public void ApplyMark(MarkBehavior newMark, GameObject player, float duration)
{
    if (currentMark == null)
    {
        //apply mark if there is no mark
        ApplyNewMark(newMark, player, duration);
    }
    else
    {
        //execute the marked dash effect if there is a mark
        MarkedDash();
    }
}

1 usage  Simon SkvaraPC +1
private void ApplyNewMark(MarkBehavior newMark, GameObject player, float duration)
{
    this.currentMark = newMark;
    this.player = player;
    this.markDuration = duration;
    this.Elapsed = 0;
    this.playerHealth = player.GetComponent<PlayerHealth>();

    currentMark.OnApply(enemy: gameObject, player);
}

1 usage  Simon SkvaraPC
private void MarkedDash()
{
    Debug.Log(message: "Attempted to do the marked dash", gameObject);

    if (markedDashAbility != null)
    {
        markedDashAbility.Execute(enemy: gameObject, player);
    }
}
```

The EnemyMarkHandler is where most of the mechanics happen. It stores the current mark and calls on the method that all marks have. And here come the abstract Scriptable objects

```
public abstract class MarkBehavior : ScriptableObject, IAbility
{
    public string markName;  Changed in 0+ assets
    [TextArea(1, 3)]
    public string markDescription;  Changed in 0+ assets
    public Sprite markIcon;  Serializable

    public GameObject markPrefab;  Changed in 0+ assets

    0+3 usages  Simon SkvaraPC
    public string AbilityName => markName;
    0+2 usages  Simon SkvaraPC
    public string AbilityDescription => markDescription;
    0+2 usages  Simon SkvaraPC
    public Sprite AbilityIcon => markIcon;


    /// <summary>
    /// Called when the mark is applied
    /// </summary>
    /// <param name="enemy">The enemy the mark is on</param>
    /// <param name="player">The player that applied the mark</param>
    1 usage  2 overrides  Simon SkvaraNB
    public abstract void OnApply(GameObject enemy, GameObject player);


    /// <summary>
    /// Execute the mark behavior as if it was Update
    /// </summary>
    /// <param name="enemy">The enemy the mark is on</param>
    /// <param name="deltaTime">Time.deltaTime</param>
    Frequently called  1 usage  2 overrides  Simon SkvaraNB
    public abstract void OnUpdate(GameObject enemy, float deltaTime);


    /// <summary>
    /// Called when the mark expires
    /// </summary>
    /// <param name="enemy">The enemy the mark is on</param>
    /// <param name="player">The player that applied the mark</param>
    Frequently called  1 usage  2 overrides  Simon SkvaraNB
    public abstract void OnExpire(GameObject enemy, GameObject player);
```

Here is the MarkBehavior, it stores abstract methods that each ability will have. Since I want

it as flexible as possible I added three methods. They each function as if it was a Start method, Update method and the OnDestroy basically. I pass necessary variables in there but also sometimes the player if it is ever needed.

Here is an example of the Default Mark and how it works.

```
public class DefaultMarkBehavior : MarkBehavior
{
    public float damagePerTick;  "20"
    public float tickInterval;  "2"

    private float timer;

    private GameObject activeMark;
    private Animator animator;

    0+1 usages  Simon SkvaraPC +1
    public override void OnApply(GameObject enemy, GameObject player)
    {
        Debug.Log(message:$"{markName} was applied to {enemy.name}", enemy);

        activeMark = AttachMarkToEnemy(enemy);

        enemy.GetComponent<EnemyMarkHandler>().activeMark = activeMark;

        animator = activeMark.GetComponent<Animator>();

    }

    Frequently called  0+1 usages  Simon SkvaraNB
    public override void OnUpdate(GameObject enemy, float deltaTime)
    {
        timer += deltaTime;
        if (timer >= tickInterval)
        {
            enemy.GetComponent<EnemyHealth>().TakeDamage(damagePerTick);
            timer = 0;
        }
    }

    Frequently called  0+1 usages  Simon SkvaraNB +1
    public override void OnExpire(GameObject enemy, GameObject player)
    {
        Debug.Log(message:$"{markName} expired on {enemy.name}", enemy);

        //Destroy(activeMark);
    }
}
```

I also made the MarkedDash, that is significantly simpler.

```
public abstract class MarkedDash : ScriptableObject, IAbility
{
    public string abilityName;  Changed in 0+ assets
    [TextArea(1, 3)]
    public string abilityDescription;  Changed in 0+ assets
    public Sprite abilityIcon;  Serializable

    0+3 usages  Simon SkvaraPC
    public string AbilityName => abilityName;
    0+2 usages  Simon SkvaraPC
    public string AbilityDescription => abilityDescription;
    0+2 usages  Simon SkvaraPC
    public Sprite AbilityIcon => abilityIcon;

    1 usage  1 override  Simon SkvaraNB
    public abstract void Execute(GameObject enemy, GameObject player);

    0+1 usages  Simon SkvaraPC
    public void AssignToPlayer(PlayerUpgradesManager manager)
    {
        manager.equippedMarkedDash = this;
    }
}
```

Over all I am extremely happy that I managed to do this and that it works. My head almost exploded while making this system but it paid off.

Week 4 (26.11. >)

I added two new ability types and that is a OnDash (when player dashes) and a Passive. Here are the scripts

```
public abstract class OnDash : ScriptableObject, IAbility
{
    public string abilityName;  ☢ Changed in 0+ assets
    [TextArea(1, 3)]
    public string abilityDescription;  ☢ Changed in 0+ assets
    public Sprite abilityIcon;  ☢ Serializable

    0+3 usages
    public string AbilityName => abilityName;
    0+2 usages
    public string AbilityDescription => abilityDescription;
    0+2 usages
    public Sprite AbilityIcon => abilityIcon;


    1 usage  1 override
    public abstract void Execute(GameObject player);

    0+1 usages
    public void AssignToPlayer(PlayerUpgradesManager manager)
    {
        manager.equippedOnDash = this;
    }
}
```

```
public abstract class Passive : ScriptableObject, IAbility
{
    public string passiveName;  ☢ Changed in 0+ assets
    [TextArea(1, 3)]
    public string passiveDescription;  ☢ Changed in 0+ assets
    public Sprite passiveIcon;  ☢ Serializable

    0+3 usages
    public string AbilityName => passiveName;
    0+2 usages
    public string AbilityDescription => passiveDescription;
    0+2 usages
    public Sprite AbilityIcon => passiveIcon;

    /// <summary>
    /// Apply the effect of the passive
    /// </summary>
    1 usage  3 overrides
    public abstract void ApplyEffect(PlayerUpgradesManager manager);

    /// <summary>
    /// if it is a passive that modifies the player stats, use for any clean up
    /// </summary>
    1 usage  3 overrides
    public abstract void RemoveEffect(PlayerUpgradesManager manager);

    1 usage  3 overrides
    public abstract void OnHitPassive(GameObject enemy, GameObject player, PlayerUpgradesManager manager);
```

In the passive there are 3 methods because for normal stat based passives you need to somehow go back to the original state if the player gets a different passive. And the on hit passive was specially implemented for the MarkChain ability (gives the mark to any enemy within the radius of the enemy currently being marked), however it can be used for other stuff.

Now, we actually want the player to have the option to choose randomly an ability. The procedural generation of this project. Now because the scriptable objects are all of different types and I didn't want to have huge hard coded scripts to handle each case I had to find a different way. Here I found/rediscovered interfaces. I never used them so it was interesting to use, but essentially I understood they can be inherited however many times a script wants. So I created a interface that would function as a global identifier and had every ability inherit it. I then added special method AssignToPlayer that would individually handle to which ability/variable they would be assigned.
I had 2 scripts, one that is essentially just taking care of the UI, displaying the information and one that is taking care of which abilities to even display.
Here is the interface.

```
15 usages  11 inheritors  0+5 exposing APIs
public interface IAbility
{
    3 usages  4 implementations
    string AbilityName { get; }
    2 usages  4 implementations
    string AbilityDescription { get; }
    2 usages  4 implementations
    Sprite AbilityIcon { get; }

    1 usage  4 implementations
    void AssignToPlayer(PlayerUpgradesManager manager);
}
```

Example of the AssignToPlayer

```
0+1 usages
public void AssignToPlayer(PlayerUpgradesManager manager)
{
    manager.equippedMark = this;
}
```

This is the script for the UI.

```
public void Initialize(PlayerUpgradesManager manager, GameObject rewardChest, PlayerMovement movement)
{
    upgradesManager = manager;
    chest = rewardChest;
    playerMovement = movement;
    playerMovement.StopMovement();
}

/// <summary>
/// Shows the option on the selection menu
/// </summary>
/// <param name="ability1">The first random ability</param>
/// <param name="ability2">The second random ability</param>
1 usage  Simon SkvaraPC
public void ShowOptions(IAbility ability1, IAbility ability2)
{
    option1 = ability1;
    option2 = ability2;

    //set name of abilities
    option1Name.text = ability1.AbilityName;
    option2Name.text = ability2.AbilityName;

    //set description of abilities
    option1Description.text = ability1.AbilityDescription;
    option2Description.text = ability2.AbilityDescription;

    //set sprite of abilities
    option1Image.sprite = ability1.AbilityIcon;
    option2Image.sprite = ability2.AbilityIcon;

    option1Button.onClick.AddListener(call: () => SelectOption(option1));
    option2Button.onClick.AddListener(call: () => SelectOption(option2));
}
```

```
2 usages  Simon SkvaraPC  More...
private void SelectOption(IAbility selectedAbility)
{
    selectedAbility.AssignToPlayer(upgradesManager);
    Debug.Log(message:$"Chosen: {selectedAbility.AbilityName}");

    playerMovement.ResumeMovement();

    Destroy(chest);
    Destroy(gameObject);

}

1 usage  Simon SkvaraPC
public void CancelSelection()
{
    Debug.Log(message:"Canceled Selection");

    playerMovement.ResumeMovement();

    Destroy(chest);
    Destroy(gameObject);
}
```

And this is the script for choosing the abilities. It is on the chest that spawns from completing a combat encounter.

```
1 usage  Simon SkvaraPC
private void OpenChest(PlayerUpgradesManager manager, PlayerMovement movement)
{
    IAbility ability1 = GetRandomAbility(manager);
    IAbility ability2 = GetRandomAbility(manager);

    // ensuring they are not same
    while (ability1 == ability2)
    {
        ability2 = GetRandomAbility(manager);
    }

    GameObject uiInstance = Instantiate(abilitySelectionUI);
    AbilitySelection selectionScript = uiInstance.GetComponent<AbilitySelection>();

    selectionScript.Initialize(manager, gameObject, movement);
    selectionScript.ShowOptions(ability1, ability2);

}
```

```
3 usages  Simon SkvaraPC
private IAbility GetRandomAbility(PlayerUpgradesManager manager)
{
    IAbility ability = null;
    int attempts = 0;

    do
    {
        ScriptableObject randomAbility = container.abilities[Random.Range(0, container.abilities.Length)];

        if (randomAbility is IAbility validAbility && !manager.IsAbilityEquipped(validAbility))
        {
            ability = validAbility;
        }

        attempts++;
    }
    while (ability == null && attempts < 10);

    return ability;
}
```

It is very nice to find out about interfaces because thanks to them the selection works quite smoothly.

The other thing that is important for the procedural generation is the random spawn position of enemies in the combat encounters. So initially it was a square spawn area and it was really easy getting a random location inside a rectangle area, but since we went away from the square rooms the spawning became more complicated. Initially, Mike (Antonio) said he would use the mechanics he uses for the enemies to move and that we would get a random position from the waypoints that are spawned. However, he later said that is not viable anymore. Honestly, thank god Sophie said we can define an area using Polygon Colliders because I went to research and found out I can store the points of a Polygon Collider. Now I

will disclose that this following piece of code was not done by me because the final math was too much for me, but at least I understand most of it. What I am doing here is retrieving 3 random points from the collider and creating a triangle and then retrieving a random point withing that triangle.

```
private Vector2 GetRandomPosition(PolygonCollider2D collider)
{
    Vector2[] localPoints = collider.points;
    Vector2[] worldPoints = new Vector2[localPoints.Length];
    for (int i = 0; i < localPoints.Length; i++)
    {
        worldPoints[i] = collider.transform.TransformPoint(localPoints[
    }

    // a random triangle (assumes the polygon is convex)
    int randomIndex = Random.Range(0, worldPoints.Length - 2);
    Vector2 p1 = worldPoints[0];
    Vector2 p2 = worldPoints[randomIndex + 1];
    Vector2 p3 = worldPoints[randomIndex + 2];

    // random value within the triangle - Barycentric
    float u = Random.value;
    float v = Random.value;

    if (u + v > 1)
    {
        u = 1 - u;
        v = 1 - v;
    }

    Vector2 randomPoint = p1 + u * (p2 - p1) + v * (p3 - p1);
    return randomPoint;
}
```

The only downside of this is that it only works for convex polygons, but it is enough for our intentions. If we had more complicated rooms, I would have to somehow make it work with non-convex polygons.

Overall we busted our asses this week, because of the upcoming Vertical Slice deadline.

Vertical Slice deadline (3. 12.)

So today we had the Vertical Slice presentation. To say the least it was a disappointing result. Although we did a lot, the result was lacking. Although Jörg had a lot of valid criticisms it felt incredibly disappointing not having validation for the hard work we put in.
Admittedly, we are making an action game and we didn't have feedback for the action. Also

the combat feels very slow, I think that partly might be because of how the enemies move and maybe the slow bullets.
The thing is, we already knew about those, but we don't have any time to do that with all the different projects we also have to work on. I suppose we will have to downscale even more to finish this game.