Technisch Ontwerp Document

Chronicle Interactive - Development

Developers
Luko ten Brink **(Lead)**
Tahsin Kaya
Alara Ilhan
Okan Emeni
Mats de Waard

Inhoudsopgave

Inleiding

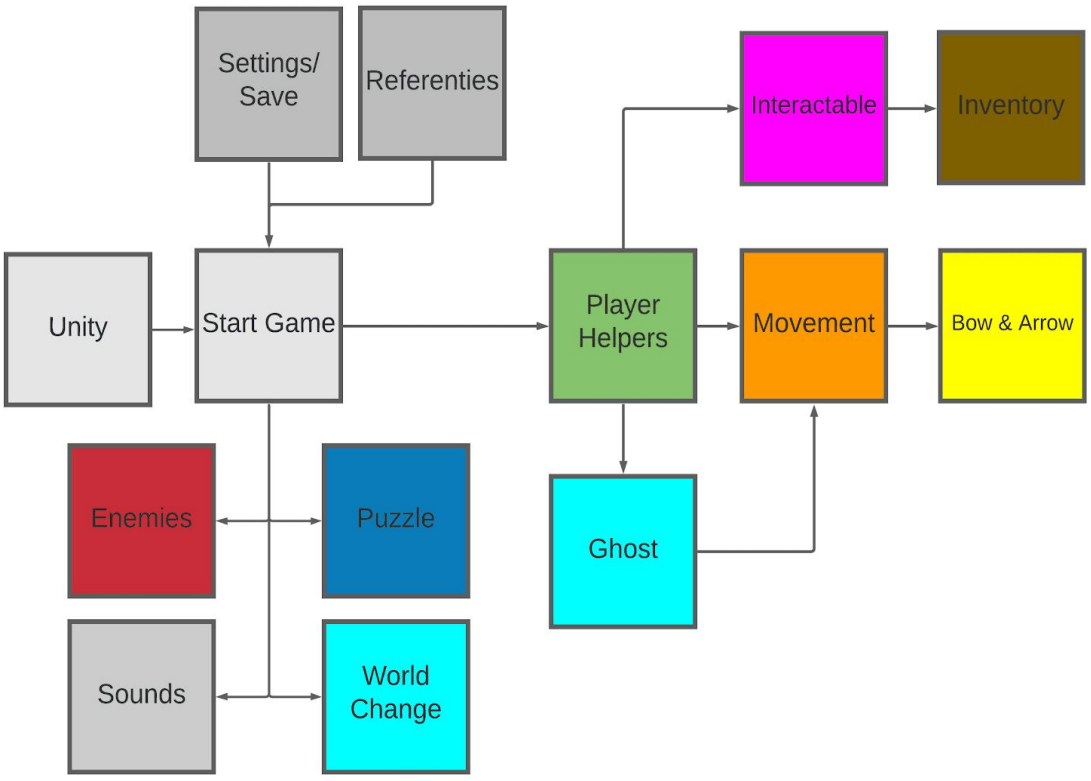
Introductie

The Shard of Kronos is gemaakt als onderdeel van de Minor Game Design & Development aan de Hogeschool Rotterdam. In dit Technisch Document worden alle belangrijke technische onderdelen benoemt voor de game. Omdat er erg veel verschillende scripts zijn in de game hebben wij ons gericht op de belangrijkste activiteiten en hebben wij de TO iets anders aangepakt. Daarnaast wordt er ook vermeld wat voor problemen we zijn tegengekomen en wat voor oplossingen er voor zijn bedacht.

Via deze link kunt men de flowchart bekijken:
https://lucid.app/lucidchart/invitations/accept/e8f16498-21ac-44b7-a1a3-39d32029d211

De flowchart is ook als een afbeelding beschikbaar met naam `Lucid Chart - Code Flow.png`

Ontwerp Overzicht



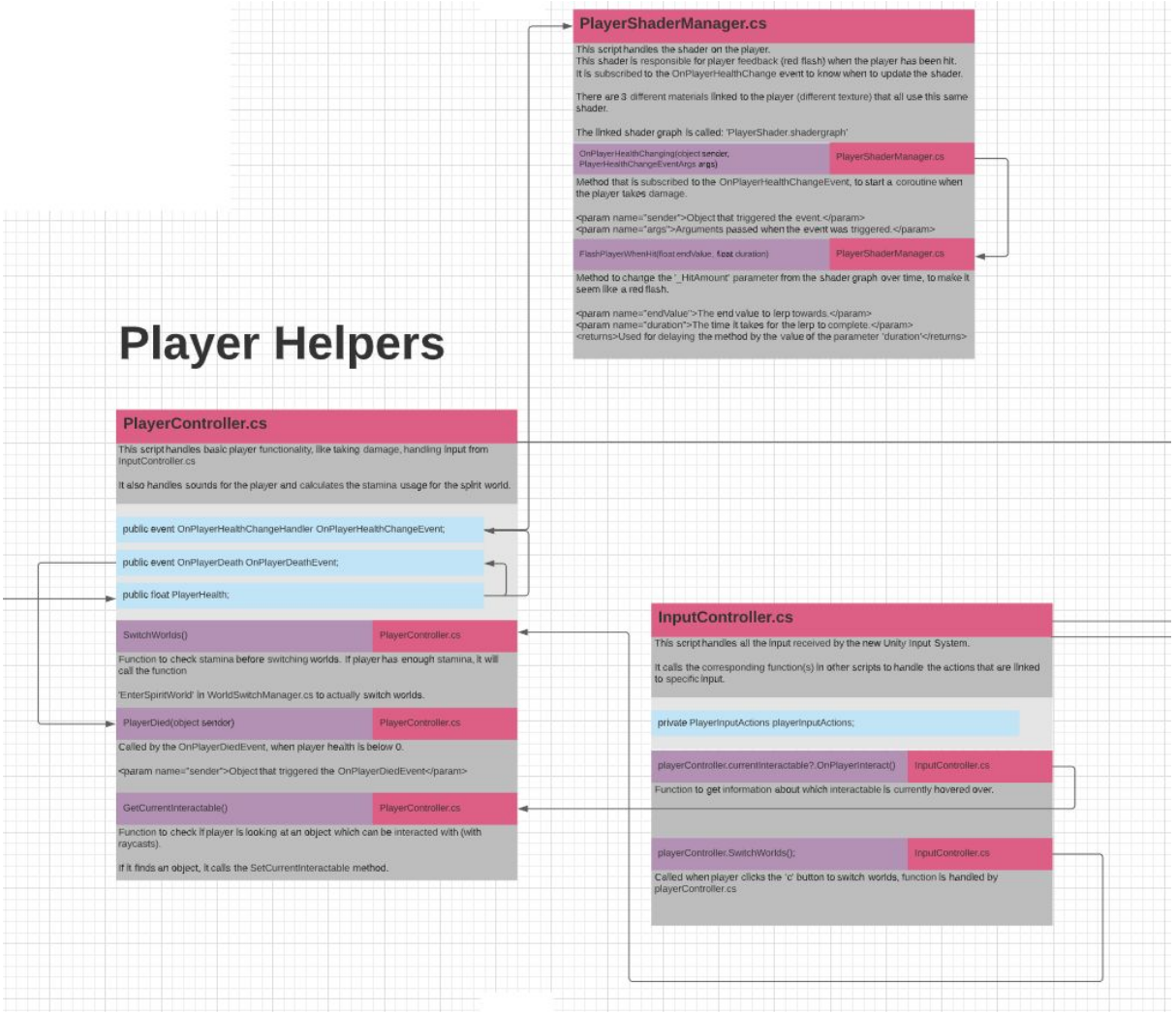
In dit document zijn de belangrijkste entiteiten/activiteiten beschreven en gegroepeerd. *Zie het schema bovenaan. Degene die een kleur heeft zijn het belangrijkst. Bij elke activiteit staat de Activiteitenbeschrijving, Theorie en algoritmen, Realisatie etc.*

Activiteit - Player Helpers

Welke scripts spelen een rol in deze activiteit?
InputController.cs - PlayerController.cs - PlayerShaderManager.cs

Wat voor gelinkte scripts zijn er?
CalculateStamina.cs - PlayerAnimatorHandler.cs - MovementHandeler.cs - PlayerHealthSettingsModel - GhostAndWorldSettingsModel - WorldSwitchManager.cs - GhostStruct.cs - PlayerStatus.cs

referentie naar flowchart



Het volledige diagram is te zien op:
https://lucid.app/lucidchart/78372da9-b978-47aa-9434-fcf67e36c00a/view#?folder_id=home&browser=icon

Korte beschrijving van de activiteit

Deze 3 scripten helpen de speler met belangrijke en grote taken. Zoals het bijhouden van de levens van de speler, het opvangen van input vanuit de speler, het weergeven van feedback wanneer de speler geraakt wordt, en het doorgeven van hoeveel de speler wilt bewegen.

Theorie

Onderzoeksvraag

Om algemene speler functies centraal te houden, hebben we een methode nodig om een aantal systemen te scheiden. De vraag is, wat is de beste manier om deze algemene speler functies op te delen, zodat het voor een ieder goed overzichtelijk blijft.

Gevonden theorie & Resultaten & conclusie

De door ons gekozen methode, is geworden om de belangrijke functies op te delen in 3 categorieën. 1. Input, 2. Movement & Health 3. Player Feedback. Daarvoor zijn dan dus ook 3 scripts gemaakt. De InputController.cs vangt alle input van de speler op met het nieuwe Unity Input Manager systeem. De PlayerController.cs regelt de movement van de speler en de health van de speler. Ten slotte zorgt de PlayerShaderManager.cs voor de speler feedback (wanneer de speler geraakt wordt).

Aan het einde van het project bleek dat dit een goede keuze was. Elke developer wist goed waar hij welke type functionaliteit kon vinden, en het was goed genoeg gescheiden dat we elkaar niet in de weg zaten wanneer we één van deze functies wilde aanpassen.

InputController.cs

Beschrijving

De InputController.cs script vangt de input van de speler op en voert de correcte functies uit. Er is gebruik gemaakt van het nieuwe Unity Input System. Hieronder is een klein stukje code te zien van hoe een input actie gelinkt zit aan de correcte uitvoerende functie:

```
playerInputActions.Actions.Jump.performed += x => Jump();

playerInputActions.Actions.SwitchWorld.performed += x =>
playerController.SwitchWorlds();

playerInputActions.Actions.ChargeBowPress.performed += x =>
ChargeBowPress();

playerInputActions.Actions.ChargeBowRelease.performed += x =>
ChargeBowRelease();
```

Sommige functies staan in het InputController.cs script zelf, en sommige worden direct in de desbetreffende scripts aangeroepen.

Theorie & Ontwerpkeuzes

Zoals eerder vermeld is ervoor gekozen om de input, playerFeedback en movement & health te splitsen. Deze ontwerp keuze was een goed idee, aangezien we geen last hebben gehad van merge conflicten wanneer mensen tegelijkertijd input probeerde toe te voegen.

PlayerController.cs

Beschrijving

De PlayerController regelt voornamelijk de speler health, paast de movement values door naar de correcte scripts en regelt de interactie met objecten.
Wanneer de playerhealth property geupdate wordt, wordt een event aangeroepen (OnPlayerHealthChangedEvent). Dit event kan op gesubscribed worden door andere classes. Ook wanneer de playerhealth < 0 wordt, is een event (OnPlayerDiedEvent) aangemaakt. Verder is gekozen om de movement van de speler te bepalen via een modulair systeem (zie hoofdstuk 'Player Movement'). De Player Controller geeft enkel nog de input waarde van de speler door aan het modulaire movement systeem.

De interactie met objecten wordt afgehandeld door de PlayerController als volgt:

```
private void GetCurrentInteractable()
    {
        interactRayCache.origin = mainCamera.transform.position;
        interactRayCache.direction = mainCamera.transform.forward;

        if (Physics.Raycast(interactRayCache, out var hit,
interactionDistance, worldSwitchManager.CurrentLayer))
        {
            var interactable =
hit.collider.gameObject.GetComponent<Interactable>();
            SetCurrentInteractable(interactable);
        }
        else SetCurrentInteractable(null);
    }

    private void SetCurrentInteractable(Interactable interactable)
    {
        if (currentInteractable != null)
currentInteractable.SetHighlight(false);
        currentInteractable = interactable;
        if (interactable != null) interactable.SetHighlight(true);
        DisplayInteractionPrompt(interactable);
```

```
    }
```

Zie hoofdstuk “Activiteit - Interaction” voor meer informatie over deze bovenstaande methods.

Theorie & Ontwerpkeuzes

Voor de player controller hadden we de keuzes voor verschillende opties. We hebben eerst overwogen om Player Health in een los script te zetten, en om de player movement volledig via de PlayerController.cs te laten gaan. Echter is hier beide vanaf gezien. Het health systeem was te klein voor zijn eigen script, en de movement wouden we modulair afhandelen, en heeft dus zijn eigen reeks scripts. Wel is er besloten om de interactie met objecten wel via de PlayerController te laten gaan, aangezien er dan niet een 4e los script op de PlayerTransform hoefde.

PlayerShaderManager.cs

Beschrijving

Het PlayerShaderManager.cs script zorgt ervoor dat de speler feedback krijgt wanneer hij geraakt wordt. Dit is gedaan met een Shader die tijdelijk de textures van de speler een rode overlay geeft zodat het een soort “blink” effect krijgt. De shader die hierbij hoort heet “PlayerShader.shadergraph”. Er is dan voor elk object in het model van de speler, die een andere texture heeft een material aangemaakt. Deze materials worden dan ingeladen in het PlayerShaderManager.cs script. Dit script is gesubscribed aan de OnPlayerHealthChangingEvent.

Wanneer dit event getriggered wordt, wordt een coroutine gestart om de shader flash in gang te zetten. Zie hieronder de coroutine:

```
private IEnumerator FlashPlayerWhenHit(float endValue, float duration)
    {
        foreach (var mat in playerMat)
        {
            mat.DOFloat(endValue, "_HitAmount", duration);
        }
        bowMat1.DOFloat(endValue, "_HitAmount", duration);
        bowMat2.DOFloat(endValue, "_HitAmount", duration);
        yield return new WaitForSeconds(duration);
        foreach (var mat in playerMat)
        {
```

```
            mat.DOFloat(0, "_HitAmount", duration);
        }
        bowMat1.DOFloat(0, "_HitAmount", duration);
        bowMat2.DOFloat(0, "_HitAmount", duration);
    }
```

Deze code zet de float genaamd '_HitAmount' op de eindwaarde 'endValue' binnen de tijd 'duration' nadat deze duur is afgelopen wordt '_HitAmount' weer naar 0 gebracht in de zelfde tijdspan.

Theorie & Ontwerpkeuzes

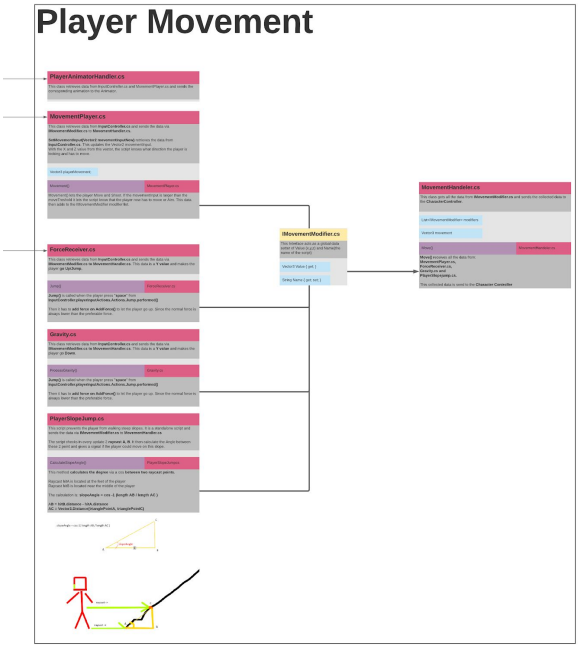
We hadden 2 opties om de speler feedback te geven van wanneer hij geraakt wordt. Door het material tijdelijk te wisselen die op het player model zit, of door een shader graph toe te voegen aan het speler model. Uiteindelijk is er gekozen voor een Shader. Dit aangezien er dan ge-fade kan worden tussen het begin en eindpunt, in plaats van een instantane verandering.

Activiteit - Player Movement

Welke scripts spelen een rol in deze activiteit?
MovementPlayer.cs - ForceReceiver.cs - Gravity.cs - PlayerSlopeJump.cs, IMovementModifier.cs, MovementHandeler.cs

Wat voor gelinkte scripts zijn er?
InputController.cs

referentie naar flowchart



Korte beschrijving van de activiteit

MovementPlayer.cs, ForceReceiver.cs, Gravity.cs en PlayerSlopeJump.cs sturen allemaal een Vector3 door naar MovementHandeler.cs. Hierdoor weet MovementHandeler.cs in wat voor directie de player moet bewegen met ***Move()***. Via de IMovementModifier.cs Interface weet de andere scripts in wat voor vorm de data moet zijn. Dit alles zorgt er voor dat de player kan bewegen.

Theorie

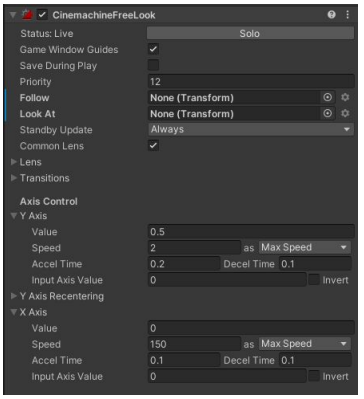
Onderzoeksvraag

De allereerste vraag was of we 1st of 3th person wilde gebruiken. Aan het begin kwam van het design team af dat we 1st person gingen gebruiken. Hiermee gebruikte we de muis input om te bepalen waar de persoon naar keek, om vervolgens in die richting in te gaan met X en Z.

Nadat we zagen dat de ghost niet helemaal duidelijk werd weergegeven in 1st person, moesten we het systeem ombouwen tot een 3th person. Daarna vroegen we ons af wat de beste manier was om een vorm van gravity te maken waarin de player soepel kan bewegen. Daarnaast moesten alle vormen van beweging, links, rechts, springen, schieten, en vallen op een manier bij elkaar komen.

Gevonden theorie, Resultaten & conclusie

Aan het begin gebruikte we het oude input systeem van Unity. Dit ging via “Horizontal”, of, “Vertical” keywords maar, we kwamen al snel er achter dat het nieuwe input system van Unity veel beter geoptimaliseerd was. Met het nieuwe systeem konden konden we makkelijk een overzicht zien wat voor inputs we gebruikte, was het compatible voor andere devices en kan dit ook makkelijk via script gemaakt worden. (Hier meer over bij InputController.cs)



Als camera movement hadden we eerst de standaard camera gebruikt van unity. Echter vonden we het niet helemaal lekker werken en besloten om te kijken naar alternatieven. Voornamelijk met de reden dat we best veel cinematische shots wilde produceren en daardoor een systeem nodig hadden dat meerdere camera’s support. Als resultaat kwamen we uit op Cinemachine. Dit uitgebreide systeem gaf de mogelijkheid om dynamische shots te creëren, het nieuwe input system was al geïntegreerd en er zaten scripts op wat ons sneller liet werken.

Nu we een manier hadden om de camera’s goed te laten bewegen hadden we een manier nodig om alle movement data op een plek te plaatsen. Na lang onderzoek kwamen we uiteindelijk uit om een interface te gebruiken en een list wat alle data ontvangt en vervolgens naar een vaste plek verstuurt.

Terugkijkend op heel het project was de input en de movement van de player soepel verlopen. Echter hadden we best wel veel problemen om de juiste settings te krijgen in de Cinemachine asset. Omdat we deze asset niet helemaal zelf hadden gemaakt, was het

moeilijk om te zien waar het probleem lag. Hierdoor is de sensitiviteit van de muis soms te hoog of te laag.

MovementHandler.cs

Beschrijving

Deze class is de ontvanger van alle verschillende x,y,z waardes en zorgt er voor dat speler de juiste kant opgaat.

Theorie & Ontwerpkeuzes

```
/// This Method receives all the different data:
/// MovementPlayer.cs - > The X and Z value.
/// ForceReceiver.cs - > The Y value.
/// PlayerSlopejump.cs - > The Y value.
/// Gravity.cs - > The Y value.
private readonly List<IMovementModifier> modifiers = new
List<IMovementModifier>();

public void AddModifier(IMovementModifier modifier) =>
modifiers.Add(modifier);
public void RemoveModifier(IMovementModifier modifier) =>
modifiers.Remove(modifier);
```

In de List<IMovementModifier> zitten alle (nieuwe) verschillende waardes van de andere scripts, bijvoorbeeld (x:0,y:0,z:2 : naar rechts) , (x:0,y:2,z:0 : omhoog).

Deze waardes worden in de **Move()** methode uitgelezen:

```
private void Move()
{
   if (changingPlayerPosition == false)
   {
       Vector3 movement = Vector3.zero;
       foreach (IMovementModifier modifier in modifiers)
       {
           movement += modifier.Value;
           if (modifier.Name == "DEV.Scripts.Input.MovementPlayer" &&
disableWasdMovement)
           {
               movement -= modifier.Value;
           }
       }

       characterController.Move(movement * Time.fixedDeltaTime);
   }
```

```
}
```

Via de foreach functie checkt het of er waardes in de List zitten, zo ja, dan verstuurt hij de ontvangen waarde `movement += modifier.Value` naar `characterController.Move(movement * Time.fixedDeltaTime);`

Aan het begin gebruikte we voor elk apart systeem, bewegen, aimen etc, een ander systeem. Dit werd uiteindelijk erg onoverzichtelijk en besloten toen om naar een ander alternatief te kijken, met dit als resultaat. De theorie is best makkelijk om te begrijpen en gebruiken. Hierdoor was het erg fijn om via deze manier te werken.

MovementPlayer.cs

Beschrijving

Deze class krijgt input van InputController.cs en stuurt de aangemaakte data naar MovementHandler.cs. Het bestuurt de beweging en het schieten van de player.

Theorie & Ontwerpkeuzes

```
public void SetMovementInput(Vector2 movementInputNew)
{
   movementInput = movementInputNew;
}
```

`movementInput` is de globale vector2 en krijgt een X & Y value van InputController.cs

Movement() geeft de waardes door in wat voor directie de speler moet bewegen.

Wanneer de player aimed, voert hij deze actie uit.

```
if (isChargingBow)
{
   if (movementInput.magnitude >= moveTreshold)
   {
       Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) *
Vector3.forward;
       Value = moveDir.normalized * curForwardSpeed / 4;
   }
   else
   {
       Value = Vector3.zero;
   }
}
```

`Vector3 moveDir` berekent in wat voor directie de player heen wilt. `curForwardSpeed` is de snelheid waarin de player kan bewegen. Wanneer de player niet aimed, roept hij het volgende aan:

```
else if (movementInput.magnitude >= moveTreshold)
{
   calculateStamina.state = CalculateStamina.StateEnum.RUN;
   transform.rotation = Quaternion.Euler(0f, angle, 0f);
   Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) *
Vector3.forward;
   Value = moveDir.normalized * curForwardSpeed;
}
else
{
   calculateStamina.state = CalculateStamina.StateEnum.IDLE;
   Value = Vector3.zero;
}
```

Hierin is het belangrijkste om te weten dat hierin de state wordt gedefinieerd *(wat nodig is voor CalculateStamina.cs)*. Verder geeft dit precies dezelfde data aan wanneer de player aimed, echter wordt de `curForwardSpeed` door 4 gedeeld wanneer de player aimed.

Via de `Value = moveDir.normalized * curForwardSpeed` wordt uiteindelijk de waarde doorgestuurd naar MovementHandler.cs.

We hebben redelijk wat verschillende methodes gebruikt om de player soepel te laten bewegen. Omdat we de nieuwe Unity Input System gebruikte waren er niet heel veel tutorials te vinden over het onderwerp en moesten we diep in de theorie duiken. Als eerste ontwerp hadden we een systeem bedacht dat de player laat bewegen maar dit kon niet aangepast worden als we bijvoorbeeld gravity wilde toevoegen. Daarnaast werd de value constant gecheckt, waardoor bijvoorbeeld de waardes elke keer 0,0,0 of 1,0,1 bleef.

PlayerAnimationHandler.cs

Beschrijving

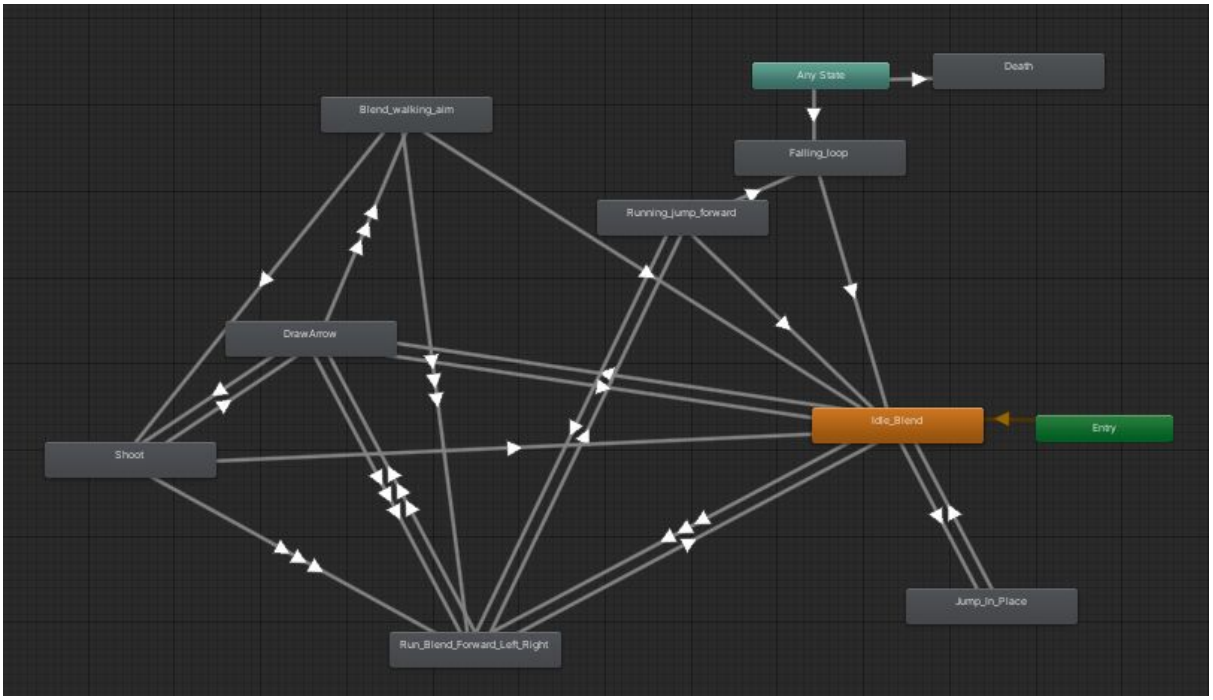
Het script zorgt er voor dat je juiste animaties worden afgespeeld en krijgt input van de InputController.cs en MovementHandler.cs. In het script worden 3 willekeurige idle animaties afgespeeld en de Animator component stuurt verschillende methodes aan.

Theorie & Ontwerpkeuzes

Het meeste spreekt voor zichzelf. In het script staan een aantal methodes die worden opgeroepen door Animator Controller:

```
...
public void AnimEvent_AddJumpForce()
{
```

```
    forceReceiver.Jump();
}

public void AnimEvent_DrawArrowCompleted()
{
    drawArrowCompleted = true;
}

public void AnimEvent_ResetJumpBoolean()
{
    pressedJump = false;
    isJumpingInPlace = false;
    movementHandeler.disableWasdMovement = false;
}
...
```



ForceReceiver.cs

Beschrijving

Via dit script kan de player omhoog en omlaag springen

Theorie & Ontwerpkeuzes

In ForceReceiver.cs krijgt **Jump()** een input wanneer de speler op de spatiebalk klikt.

```
public void Jump()
{
    if (characterController.isGrounded)
    {
        calculateStamina.state = CalculateStamina.StateEnum.JUMP;

        Vector3 force = new Vector3(0, physicsModel.jumpForce, 0);
        AddForce(force);
    }
}
```

Wanneer de player op de grond staat (wat de component, CharcterController, checkt) en spatie indrukt, krijg de Addforce(force) de force van de jump terug. Deze force wordt bepaald door de `physicsModel.jumpForce` setting.

```
public void AddForce(Vector3 force)
{
    Vector3 forceCalc = force / physicsModel.mass;
    Value += Vector3.Lerp(forceCalc, Vector3.zero, 1 * Time.deltaTime);
}
```

Hierin wordt de uiteindelijke force (jump) bepaald. `Vector3 forceCalc = force / physicsModel.mass;` De force is de power van de jump en wordt gedeeld door de aangegeven mass van de player. Dan wordt via de lerp berekent hoeveel force de player heeft. Als voorbeeld, de forceCalc ; 10 / 2 = 5. De player springt dat met de waarde 5 omhoog. Deze waarde wordt dan gelerpt naar 0, waardoor de player weer naar beneden valt aangezien de waarde naar 0 wordt gezet.

Gravity.cs

Beschrijving

Dit script checkt continue of de player op vaste grond staat of in de lucht. Als de player in de lucht staat wordt hij naar beneden geplaatst.

Theorie & Ontwerpkeuzes

```
private void ProcesGravity()
{
   //If we are on the ground, we have groundedPullMagnitude, so we stay
on the ramp.
   if (characterController.isGrounded)
   {
       Value = new Vector3(Value.x, -physicsModel.groundedPullMagnitude,
Value.z);
   }
   //We just jumped, resets value to 0 , since groundedPullMagnitude is
really low in the -
   else if (wasGroundedLastFrame)
   {
       Value = Vector3.zero;
   }
   //If we are falling
   else
   {
       Value = new Vector3(Value.x, Value.y + gravityMagnitude *
Time.deltaTime, Value.z);
   }

   wasGroundedLastFrame = characterController.isGrounded;
}
```

In kort, via de line:

```
  if (characterController.isGrounded)
   {
       Value = new Vector3(Value.x, -physicsModel.groundedPullMagnitude,
Value.z);
   }
```

Wordt er gekeken of de player nog op de grond is. Zo ja, dan wordt de player naar beneden “geduwd”. Dit werkt net zoals echte zwaartekracht.

```
 else
   {
       Value = new Vector3(Value.x, Value.y + gravityMagnitude *
Time.deltaTime, Value.z);
   }
```

Als de CharacterController niet meer aangeeft dat de player op de grond staat, wordt deze functie uit geroepen. Dit geeft aan hoe snel de player naar beneden moet vallen.

Dit script heeft geen verdere aanpassingen gehad.

PlayerSlopeJump.cs

Beschrijving

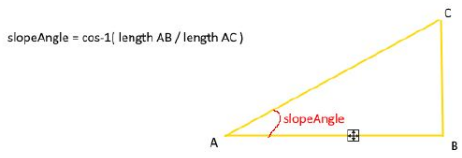
Dit script zorgt ervoor dat de player geen steile randen kan beklimmen.

Theorie & Ontwerpkeuzes

```
private (float slope, GameObject obj) CalculateSlopeAngle()
{
   RaycastHit hitA;
   RaycastHit hitB;

   if (!Physics.Raycast(transform.position, transform.forward, out hitA,
2)) return (0, null);
   if (!(Physics.Raycast(transform.position + Vector3.up,
transform.forward, out hitB, 2) &&
         hitB.collider == hitA.collider)) return (0, null);

   Vector3 trianglePointA = hitA.point;
   Vector3 trianglePointC = hitB.point;

   float distAB = hitB.distance - hitA.distance;
   float distAC = Vector3.Distance(trianglePointA, trianglePointC);
   return (Mathf.Acos(distAB / distAC) * Mathf.Rad2Deg,
hitA.collider.gameObject);
}
```

Deze methode berekent de graden van een platform via een cosinus tussen 2 raycast punten. Raycast hitA bevindt zich op de voeten van de speler en raycast hitB is op de middel. De calculate is:

AB = hitB.distance - hitA.distance
AC = Vector3.Distance(trianglePointA, trianglePointC)



slopeAngle = cos -1 (length AB / length AC)
A**B = hitB.distance - hitA.distance**
AC = Vector3.Distance(trianglePointA, trianglePointC)
Wanneer de uitkomst bijvoorbeeld hoger dan 30 is, belemmert het de player om te bewegen. Hierdoor kan de player niet meer dan 30* graden omhoog lopen

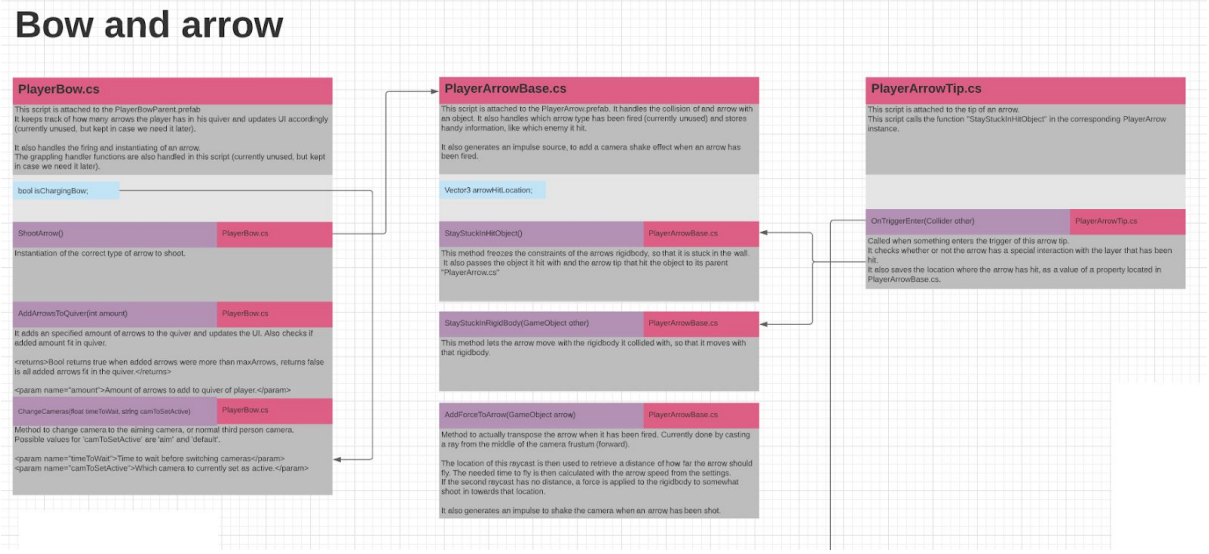
Dit script is later in het project toegevoegd als hulpmiddel zodat de player niet overal bij kan.

Activiteit - Bow and Arrow (aanvallen)

Welke scripts spelen een rol in deze activiteit?
PlayerBow.cs - PlayerArrowBase.cs - PlayerArrowTip.cs

Wat voor gelinkte scripts zijn er?
BowAndArrowSettingsModel - WorldSwitchManager.cs - PlayerBow.cs

referentie naar flowchart



Het volledige diagram is te zien op:
https://lucid.app/lucidchart/78372da9-b978-47aa-9434-fcf67e36c00a/view#?folder_id=home&browser=icon

Korte beschrijving van de activiteit

De speler moet kunnen aanvallen. Hiervoor is gekozen om een pijl en boog als wapen te gebruiken. De speler kan de rechtermuisknop ingedrukt houden om zijn boog ‘op te laden’ en dan de pijl te schieten door de linkermuisknop in te drukken (voordat de rechtermuisknop is losgelaten). De camera verandert ook van aanzicht wanneer de speler de rechtermuisknop ingedrukt houdt, net als de muisgevoeligheid. Dit is zodat de speler gemakkelijker kan richten.

Theorie

Onderzoeksvraag

Het belangrijkste om te weten is, hoe kan de gebruiker vechten? We hebben gekozen om een simpel, draw, aim, shoot systeem te maken. De speler kan rechtermuisknop ingedrukt houden om in te zoomen, daarna kan hij de linkermuisknop gebruiken om te schieten.

De andere keuzes die we overwogen hebben was om de afstand van de geschoten pijl aan te passen aan de hand van een “charge” meter die volloopt zolang de speler de rechtermuisknop ingedrukt houdt. Dit bleek echter achteraf een slecht idee, omdat het richten en raken van vijanden daardoor heel inconsistent werd.

Ook is het script van de pijlen zo geschreven dat er een mogelijkheid bestaat om andere soorten pijlen te kunnen afschieten. Dit wordt reeds niet gebruikt, maar het object is opgebouwd zodat dit makkelijk geïmplementeerd kan worden.

Hetzelfde geldt voor de grapling hook functionaliteiten. We waren van mening dat we een grappling hook wilde gaan gebruiken in de game. Hiervoor is de code al geschreven, maar uiteindelijk is dit uitgezet aangezien we de animaties niet gemakkelijk aan de praat kregen. Terwijl we maar één hookpoint in de game hadden. Dus hebben we het maar volledig uitgezet.
Ten slotte is de kwestie er of we unlimited pijlen of niet wilde. Dit is momenteel opgelost door een quiver systeem toe te voegen, maar in te stellen dat de speler begint met 1000000 pijlen (per scene load) waardoor hij praktisch gezien oneindig aantal pijlen heeft. Dit kan veranderd worden wanneer bijvoorbeeld de verschillende typen pijlen geïmplementeerd worden.

Gevonden theorie & Resultaten & conclusie

De methodiek die uiteindelijk gebruikt is om te kunnen schieten is als volgt:
Op het player model zit een script “PlayerBow.cs” deze regelt de input van de speler en het instantiaten van de pijlen. Wanneer dit script een pijl instantieert met het script “PlayerArrowBase.cs” erop, vliegt er een pijl door het level. Op deze pijl zit op de punt een trigger, waar ook een script aan gekoppeld zit (PlayerArrowTip.cs). Deze trigger geeft door aan de pijl wanneer hij iets raakt. Vanuit het script op de pijl wordt dan afgehandeld wat er moet gebeuren met deze collision (blijft de pijl hangen, moet de pijl kapot, moet de pijl schade doen etc).

PlayerBow.cs

Beschrijving

Wanneer PlayerBow.cs input krijgt van InputController.cs zet hij de bool, ***isCharging*** naar true. Wanneer de bool ***isCharging*** true is, veranderd hij de camera view naar de Cinemachine Aim Camera. Wanneer **isCharging == true;** dan wordt een trigger in de player animator aangezet die de “Draw” animatie start. Wanneer deze animatie klaar is, en de speler op de linkmuisknop drukt, wordt de functie ShootArrow() uitgevoerd. Deze is als volgt:

```
public void ShootArrow()
        {
            calculateStamina.state = CalculateStamina.StateEnum.ATTACK;

            Vector3 spawnLoc = arrowSpawnLoc.position +
                               (arrowSpawnLoc.right.x *
bowAndArrowSettings.cameraOffsetWhenDrawingBow);
            ArrowType typeToSpawn = ArrowType.Default;
            if (hasSelectedSpecialArrows)
            {
                typeToSpawn = ArrowType.IceAoE;
            }


            GameObject arrowObjectToSpawn;
            switch (typeToSpawn)
            {
                case ArrowType.IceAoE:
                    arrowObjectToSpawn =
bowAndArrowSettings.iceAoEPrefab;
                    break;
                default:
                    arrowObjectToSpawn =
bowAndArrowSettings.defaultArrowPrefab;
                    break;
            }

            GameObject arrow = Instantiate(arrowObjectToSpawn, spawnLoc,
playerTransform.rotation,
                arrowSpawnParent);

arrow.GetComponent<PlayerArrowBase>().AddForceToArrow(arrow);
            curCharge = 0;

            RemoveArrowsFromQuiver(1);
            if (curArrows <= 0)
            {
```

```
                SetActiveChargeUI(false);
            }
        }
```

Deze functie haalt stamina weg bij het Spirit World stamina systeem, kiest welke pijl er geschoten moet worden (momenteel alleen nog maar type = default) en instantiate daarna de pijl. Hierna wordt er één pijl uit de quiver van de speler gehaald.

Theorie & Ontwerpkeuzes

Zoals al eerder vermeld, zijn er veel keuzes en concessies gemaakt en bedacht voor het bow & arrow systeem. Het quiver systeem werkt momenteel wel, maar staat ingesteld op 1000000 pijlen, ook de verschillende typen pijlen zijn mogelijk te gebruiken, echter is er alleen nog maar één soort pijl (de standaard pijl). Ook is het charge systeem om de pijl met verschillende forces te schieten uitgeschakeld om het aimen/schieten consistent te houden. Er is wel overwogen om inplaats van de force te varieren om de damage te varieren aan de hand van hoeveel je de boog hebt gecharged. Echter is ook deze optie van de kaart geveegd omdat we geen passende manier konden bedenken om dit aan de speler terug te koppelen.

PlayerArrowBase.cs

Beschrijving

De PlayerArrowBase class is de hoofdklasse van alle verschillende soorten pijlen. Deze class regelt alle functie en fields die van belang zijn voor elke pijl. Zo bevat het de functie om de pijlen naar de juiste eindlocatie te bewegen, en heeft het fields om bij te houden of deze pijl al eens damage heeft gedaan aan een enemy.

Theorie & Ontwerpkeuzes

De default arrow is als hoofd class geschreven om zo gemakkelijk andere pijlen te kunnen toevoegen aan het spel. Echter is dit er niet van gekomen door tijdsgebrek bij de game designers. Het ging te moeilijk worden om op een leuke manier nog verschillende soorten pijlen toe te voegen aan het spel binnen de tijd van de deadline

PlayerArrowTip.cs

Beschrijving

De tip van de arrow is een los GameObject met een collider die op trigger mode staat. Deze collider geeft door wanneer hij iets raakt, en reageert gepast op met wat hij heeft gecollide.

```
 private void OnTriggerEnter(Collider other)
        {
            Debug.Log(other.gameObject.name);
            if (other.gameObject.layer == IgnoreArrowLayer ||
other.gameObject.name == "hint1")
            {

            }
            else
            {
                playerArrowParentScript.StayStuckInHitObject();
            }

            playerArrowParentScript.ArrowHitLocation =
playerArrowParentScript.gameObject.transform.position;

            switch (other.gameObject.layer)
            {
                case ObstacleLayer:

                    if (playerArrowParentScript.ArrowType == ArrowType.IceAoE)
                    {

playerArrowParentScript.GetComponent<PlayerArrowIceAoE>().StartIceAoEEffect();
                    }

                    break;
                case EnemyLayer:
                    //playerArrowParentScript.StayStuckInHitObject();
                    break;
                case GrappleLayer:
                    playerArrowParentScript.StartGrapple(other, GrappleLayer);
                    break;

            }
        }
```

Theorie & Ontwerpkeuzes

Dit script zit enkel op de tip van een pijl, en zal dus mee geinstantieerd worden wanneer een pijl geschoten wordt. Het bovenstaande “OnTriggerEnter()” methode, bepaald wat voor actie wordt ondernomen wanneer de pijl iets raakt.

Activiteit - Referenties naar scripts

Welke scripts spelen een rol in deze activiteit?
SceneContext.cs, Settings.cs,SettingsEditor.cs

Wat voor gelinkte scripts zijn er?
WorldSwitchManager.cs - GhostCheckpointParent.cs - CalculateStamina.cs - GhostStruct.cs - PlayerController.cs - LightActivatorPillar.cs - MeteoorrExplode.cs - SpawnFracturedMeteoorr.cs - PlayerStatus.cs - RespawnManager.cs -

referentie naar flowchart
-

Korte beschrijving van de activiteit

Een Heldere beschrijving met schematische weergave van een activiteit die plaatsvindt binnen het technische product

Om scripts met elkaar te verbinden wordt in de scenecontext een referentie gemaakt. Er wordt eerst een variable aangemaakt, met als type de classnaam, in de SceneContect.cs

```
[BoxGroup("General objects")] public WorldSwitchManager
worldSwitchManager;
```

In een andere script zal er nu een variable aangemaakt moeten worden om de connectie te leggen.

```
private WorldSwitchManager worldSwitchManager;
```

Als voorbeeld nemen we de Ghosstruct. Nadat de worldSwitchManager een instance is van de scenecontext, kan er via de Ghoststruct.cs connectie worden gemaakt met de worldSwitchManager.cs.

```
worldSwitchManager = SceneContext.Instance.worldSwitchManager;
```

Theorie

Onderzoeksvraag

Hoe kunnen we ervoor zorgen dat scripts gelinkt kunnen worden met andere scripts?

Gevonden theorie

Bij het coderen worden er voornamelijk referenties gemaakt naar verschillende scripts. Om dit op een makkelijke manier aan te pakken hebben we een script aangemaakt genaamd SceneContext. Hierin maken we een variable aan met de classnaam als type waarnaar de

referentie wordt gemaakt. De scenecontext script fungeert als een singleton design pattern en zit op een gameobject.

In de eerste weken van het project hebben we gebruik gemaakt van scriptableobjects. De scriptableobject voeg je toe aan een gameobject en kunnen we zien als een design pattern. Deze wordt vervolgens toegevoegd aan een lijst en wordt daarin opgeslagen. Tenslotte kun je de connectie maken naar de script.

In Unity functioneerde deze design pattern zonder errors. Echter bij de build versie van de game is er opgemerkt dat de lijst niet werd gevuld met de game objecten die de scriptable object hadden. Dit is de belangrijkste reden waarom we zijn overgestapt naar een singleton.

SceneContext#1

Beschrijving

In de SceneContect.cs worden er public variables aangemaakt met als type de script waarnaar je het wilt connecten. Deze zijn public gemaakt omdat we in de inspector de gameobjects kunnen slepen. Om overzicht te houden zijn er boxgroups aangemaakt met de juiste namen. Deze script is geplaatst op een gameobject. De gameobject is ook een prefab en wordt in elke scene gebruikt.

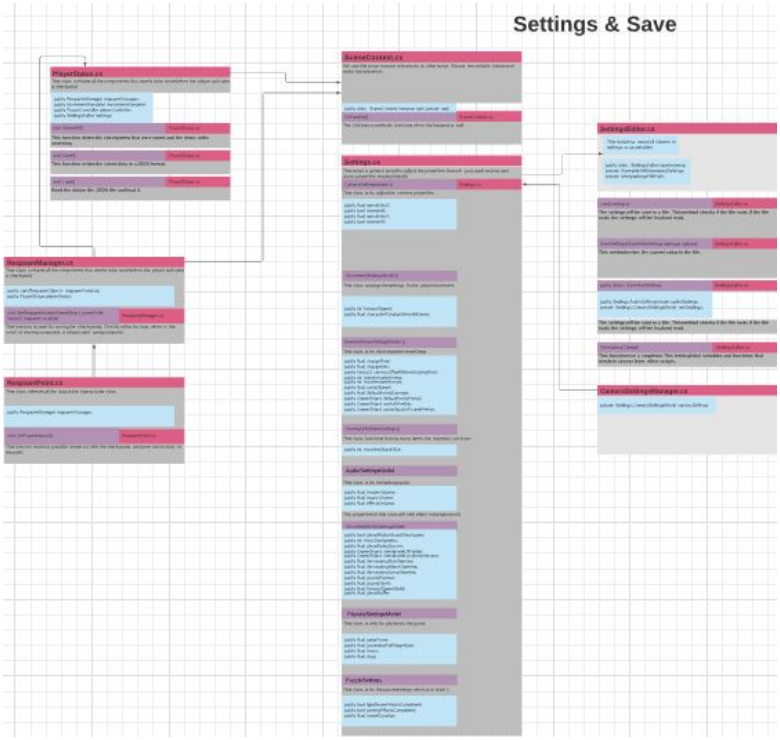
```
[BoxGroup("General objects")] public WorldSwitchManager
worldSwitchManager;
        [BoxGroup("General objects")] public Transform
objectsSpawnedByPlayer;
        [BoxGroup("General objects")] public GhostCheckpointParent
ghostCheckpointParent;
        [BoxGroup("General objects")] public Transform bulletTransform;
        [BoxGroup("General objects")]public CalculateStamina
calculateStamina;
        [BoxGroup("General objects")]public GhostStruct ghostStruct;
        [BoxGroup("General objects")] public PlayerController
playerController;
        [BoxGroup("General objects")] public LightActivatorPillar
lightActivatorPillar;
        [BoxGroup("General objects")] public MeteoorrExplode
meteoorrExplode;
        [BoxGroup("General objects")] public SpawnFracturedMeteoorr
spawnFracturedMeteoorr;
        [BoxGroup("Player")] public PlayerController playerManager;
        [BoxGroup("Player")] public Transform playerTransform;
        [BoxGroup("Player")] public GameObject playerBowParent;
        [BoxGroup("Player")] public PlayerStatus playerStatus;
        [BoxGroup("Player")] public GameObject ghostObject;
```

Activiteit - Opslaan van settings

Welke scripts spelen een rol in deze activiteit?
Settings.cs,SettingsEditor.cs

Wat voor gelinkte scripts zijn er?
LevelAudioManager.cs - PlayerBow.cs - PlayerArrowBase.cs - EnemyBase.cs - GhostCheckpointParent.cs - GhostStruct.cs - WorldSwitchManager.cs - CalculateStamina.cs - CameraSettingsManager.cs - ForceReceiver.cs - MovementPlayer.cs - PlayerController.cs - PlayerStatus.cs - AudioSettings.cs

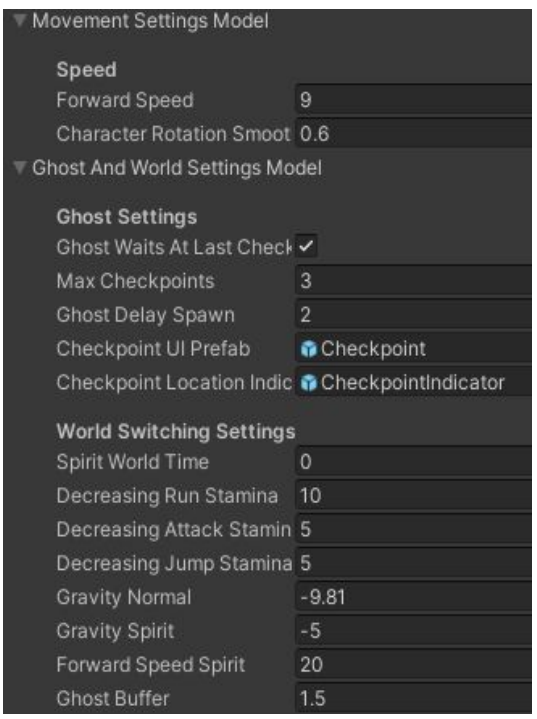
referentie naar flowchart

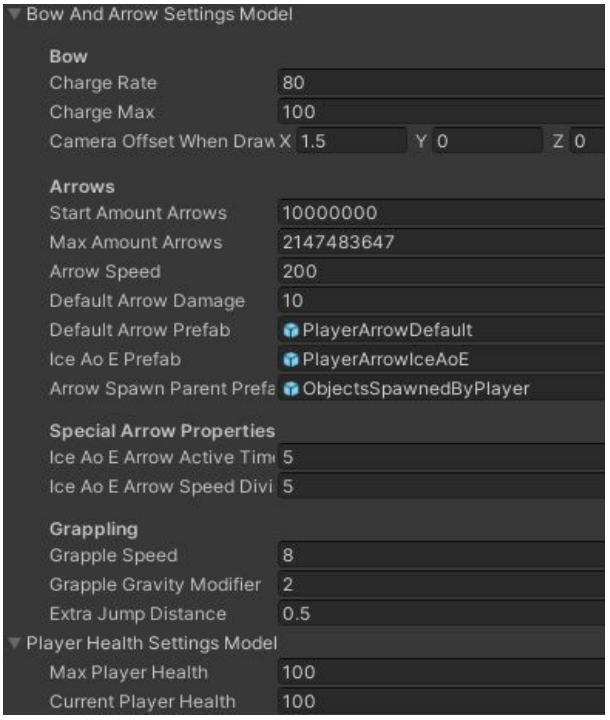


Korte beschrijving van de activiteit

Een Heldere beschrijving met schematische weergave van een activiteit die plaatsvindt binnen het technische product

Er is een prefab genaamd “Settings”. Hierin worden alle settings in opgeslagen en hebben direct effect op de game. Deze variables zijn altijd aanpasbaar via de inspector.





.

Theorie

Onderzoeksvraag

Hoe kunnen we ervoor zorgen dat er via 1 script de variables aangepast kunnen worden van bepaalde classes?

Gevonden theorie

Deze script werkt als een singleton. Het voordeel hiervan is dat je de variables van bepaalde classes tot 1 omgeving kan opslaan en genereren. Zo zijn alle variables makkelijk aanspreekbaar en zijn makkelijk aan te passen door de developers en de designers.

We hebben gekozen voor deze design pattern omdat het op een krachtige manier werkt. Daarbij is het ook overzichtelijk en in een oogopslag te zien welke variables bij welke scripts horen.

Settings#1

Het opslaan van de settings voor elk onderdeel wordt in deze script gedaan. Er wordt een class aangemaakt met de properties. Deze class is static gemaakt omdat deze wordt aangeroepen in de SettingsEditor.cs. Hierdoor zal de singleton functie in zijn werk gaan.

```
public static class Settings
{

    #region - Player -

    [Serializable]
    public class CameraSettingsModel
    {
        [Header("Camera Settings")]
        public float sensitivityX = 150;
        public bool invertedX = false;
        public float sensitivityY = 2;
        public bool invertedY = true;
        public float aimCamXDivider = 4;
        public float aimCamYDivider = 8;

    }

    [Serializable]
    public class MovementSettingsModel
    {
        [Header("Speed")]
        public float forwardSpeed;
        public float characterRotationSmoothDamp = 0.6f;
    }
```

Theorie & Ontwerpkeuzes

Settings.cs zit op een gameobject, wat een prefab is. De prefab heeft effect op alle scenes in de game.

SettingsEditor#2

Deze script werkt als een singleton. Settings.cs is verbonden aan de deze script. SettingsEditor is static gemaakt omdat het eenmalig in elke script die het nodig heeft om de properties op te halen, wordt aangeroepen. De volgende settingsmodels van Settings.cs zijn verbonden met deze script.

```
 public Settings.CameraSettingsModel cameraSettingsModel;
    public Settings.MovementSettingsModel movementSettingsModel;
    public Settings.GhostAndWorldSettingsModel
ghostAndWorldSettingsModel;
    public Settings.BowAndArrowSettingsModel bowAndArrowSettingsModel;
    public Settings.PlayerHealthSettingsModel playerHealthSettingsModel;
    public Settings.AudioSettingsModel audioSettingsModel;
    public Settings.PhysicsSettingsModel physicsSettingsModel;
    public Settings.PuzzleSettings puzzleSettings;
    public Settings.InventoryAndItemSettings inventorySettings;
    public Settings.EnemySettingsModel enemySettingsModel;


    private static SettingsEditor classInstance;
```

Hieronder wordt de code getoond hoe de singleton werkt. Als de classInstance null is dan wordt de methode OnInstanceCreate aanroepen en anders de classInstance. De settings gameobject wordt vervolgens opgeslagen als een prefab en geladen vanuit de Resources folder. Daarna wordt het geinstantiate.

```
#region Singleton Logic

    public static SettingsEditor Instance => classInstance == null ?
OnInstanceCreate() : classInstance;

    private static SettingsEditor OnInstanceCreate()
    {
        var resName = "Settings";
        var prefab = Resources.Load<GameObject>(resName);
        var gameObjectInstance = Instantiate(prefab);
        gameObjectInstance.name = resName;
        classInstance =
gameObjectInstance.GetComponent<SettingsEditor>();
        return classInstance;
    }
```

Theorie&Ontwerpkeuzes

Na een aantal weken hadden door dat we teveel properties hadden waardoor we steeds in elke script de variables moesten gaan aanpassen wat veel tijd kostte en niet overzichtelijk was. Daarna zijn we op het idee gekomen om een singleton te gebruiken. Alle variables worden op een plek opgeslagen en is makkelijk aanpasbaar.

Activiteit - SaveSystem

Welke scripts spelen een rol in deze activiteit?
PlayerStatus.cs, RespawnManager.cs, RespawnPoints.cs

Wat voor gelinkte scripts zijn er?
PlayerStatus.cs - RespawnManager.cs - RespawnPoints.cs - Interactable.cs - Ghoststruct.cs

referentie naar flowchart
...

Korte beschrijving van de activiteit

Een Heldere beschrijving met schematische weergave van een activiteit die plaatsvindt binnen het technische product

De speler kan met de toets “V” een checkpoint zetten. Er zijn in totaal 3 checkpoints die gezet kunnen worden in de spiritwereld. Wanneer de checkpoint gezet wordt, gaat de savesystem in actie. Om terug te spawnen naar de checkpiont die gezet is, gebruik je de toets ”B”. Hierdoor blijven er 2 checkpoints over. Tegelijkertijd kijkt de savesystem ook in welke scene de player zit.

PlayerStatus.cs haalt de data op waar de checkpoints worden opgeslagen in een list. Dit wordt opgeslagen in de RespawnManager.cs. Deze list wordt opgeslagen in de variable saveablelist en wordt vervolgens geschreven naar een JSON bestand.

Theorie

Onderzoeksvraag

Hoe kunnen we ervoor zorgen dat bepaalde acties worden opgeslagen tijdens het spelen van de game, zoals het zetten van de checkpoints.

Gevonden theorie

Hiervoor is er een save system geïmplementeerd. Eerst worden de gegevens opgehaald die bewaard moeten worden, daarna worden ze de variable saveableList opgeslagen. Vervolgens wordt die variable geschreven en opgeslagen als een JSON file in een maplocatie.

PlayerStatus#1

Beschrijving

In SaveFile struct geven we in de constructor mee wat er gesaved moet worden. De checkpoints die gezet worden, komen vanuit RespawnObject.cs. Dit wordt opgeslagen in de respawnPontListStruct.

```
public struct SaveFile
        {

            public string currentScene;
            public List<RespawnObject> respawnPointListStruct;

            public SaveFile( RespawnManager respawnManager)
            {
                currentScene = SceneManager.GetActiveScene().path;
                respawnPointListStruct =
respawnManager.respawnPointList;
            }
        }
```

```
private SaveFile saveableList;
```

De saveableList wordt hier opgeslagen en geschreven naar een path als JSON bestand.

```
 public void Save()
        {

            Debug.Log("Saving...");
```

```
            saveableList = new SaveFile(respawnManager);

            string path = Path.Combine(Application.persistentDataPath,
saveFileName); //Where to save?

            var json = JsonUtility.ToJson(saveableList, true);
            //Debug.LogWarning($"Save file JSON:\n{json}" );
            Debug.Log(json);

            File.WriteAllText(path, json);
            //Debug.Log("save" + json);
        }
```

De DeleteAll functie zorgt ervoor dat de list van de respawnpoints die worden aangemaakt in RespawnManager.cs worden verwijderd.

```
respawnManager.respawnPointList.Clear();
```

De load functie controleert of het bestand bestaat, waar de data in opgeslagen is. Als het bestand bestaat, wordt de data gelezen en geladen, in tegenstelling zal hij de vorige data lezen en laden.

```
            if (File.Exists(string.Concat(path)))
            {
                var json = File.ReadAllText(path);
                var obj = JsonUtility.FromJson<SaveFile>(json);
                //                Debug.Log(json);
            }
            else
            {
                Save();
                Load();
            }
```

Theorie & Ontwerpkeuzes

We hebben voor een savesystem gekozen omdat het makkelijk uitbreidbaar is en als er in de toekomst andere data moet worden opgeslagen kan het gebruikt worden.

RespawnManager#2

Beschrijving

Deze class houdt bij welke properties toegevoegd moet worden wanneer de player een checkpoint activeert. Er wordt een nieuwe instantie gemaakt van de class en vervolgens wordt dit toegevoegd in de respawnPointList.

```
   public class RespawnObject
    {
        public string sceneName;
        public int index;
        public GameObject respawnObject;
        public bool isActivated;
        public Vector3 vector3Transform;
    }
```

```
var obj = new RespawnObject
                {
                    sceneName = SceneManager.GetActiveScene().name,
                    index = var.GetSiblingIndex(),
                    vector3Transform =
var.transform.GetChild(0).transform.position,
                    respawnObject = var.gameObject,
                    isActivated = false
                };
                respawnPointList.Add(obj);
```

RespawnPoint#3

Beschrijving

De OnPlayerInteract functie regelt dat er interactie(input) gemaakt kan worden met de checkpoints die gezet worden.

```
 public override void OnPlayerInteract()
    {
        respawnManager.SetRespawnActive(this.gameObject,
respawnLocation);
```

```
    }
```

Theorie & Ontwerpkeuzes

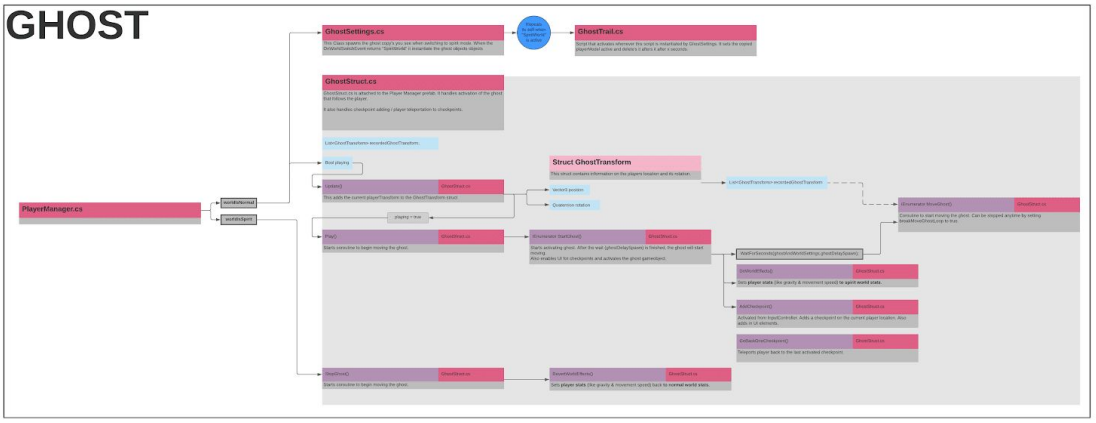
Het plaatsen van de checkpoints en de mogelijkheid om terug te spawnen op de checkpoints wordt geregeld in GhostStruct.cs. Om de inputs hiervoor te regelen zijn twee methodes aangemaakt welke worden aangeroepen in de inputcontroller.

Activiteit - Ghost

Welke scripts spelen een rol in deze activiteit?
GhostStruct.cs - GhostSettings.cs - Ghost Trail.cs - ShockWaveEffect.cs

Wat voor gelinkte scripts zijn er?
PlayerController.cs - MovementPlayer.cs - InputController.cs

referentie naar flowchart

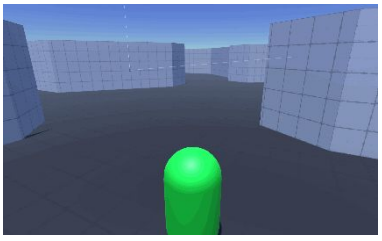


Korte beschrijving van de activiteit

De player heeft een vorm van herkenning nodig om te weten wanneer hij in de “Spirit world” zit en dat hij een ghost achterlaat. In PlayerManager.cs wordt er aangegeven in welke fase de player zit. Als de player in de “Normale fase” zit, oftewel, ***worldIsNormal*** *en hij klikt op “C”,* activeert hij deze activiteit. Deze activiteit zorgt er dan voor dat:

- De player een ghost achterlaat op de laatste positie van de model en locatie van de player.
- De player in staat is om terug te switchen naar de normale wereld.
- De player in staat is om checkpoints te maken in de spirit mode en hier op terug te teleporteren wanneer hij switch naar de normale wereld.

Theorie



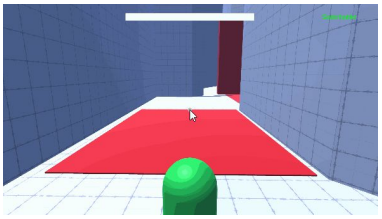
Onderzoeksvraag

De belangrijkste vraag was eerst hoe we de ghost achter ons konden laten bewegen. De player moest namelijk een kopie van zichzelf zien. Deze kopie gaaf aan waar hij terug naar teleporteerde als hij terug ging naar de normale wereld.

De volgende stap was er voor te zorgen dat de ghost maar een X aantal seconden de player mag achtervolgen voordat

de player terug wordt geteleporteerd. Daarnaast moest de player in staat zijn om checkpoints te maken.

Wanneer dit werkte, moesten sommigen key objecten die rondom de player staan, veranderen in de spirit mode. Daarnaast moest er ook visueel worden aangegeven in welke wereld de player zit.



Gevonden theorie & Resultaten & conclusie

```
public struct GhostTransform
{
    public Vector3 position;
    public Quaternion rotation;

    Mist3rWh0 +1
    public GhostTransform(Transform transform)
    {
        position = transform.position;
        rotation = transform.rotation;
    }
}
```

Voor de ghost kopieën moest er een systeem worden bedacht waarin de player zijn position en rotation wordt opgeslagen om de X seconden. De eerste methode was om de positite en rotatie op te slaan in een Struct dat dan weer wordt opgeslagen in een List. *(zie GhostStruct.cs)* In deze List zit dus elke keer een blokje van informatie over de locatie en rotatie van de player. Dit werkte best wel goed en was qua performance niet heel erg intensief.

Echter kwamen we er achter dat een volledig model kopiëren lastig was. Vooral om de juiste houding terug te krijgen van de player. Hierdoor moesten we switchen naar een andere methode. In plaats van de locatie en rotatie te kopiëren, wordt de Mesh van de player ge-instantiate. Met voorname de polySurface waarin de gegevens staan wat de houding van de player is.

Omdat we nu een systeem hadden waarin stond wat de positie van de player was en de houding, kon de checkpoint systeem hier makkelijk op aansluiten. Wanneer de player op V klikt kijkt het script naar welke positie het laatst is opgeslagen en slaat dit op als een mogelijke respawn point.

Als laatste moesten wij een manier verzinnen om objecten te laten verdwijnen en duidelijk te maken dat de player switch van wereld. Dit is uitgevoerd door met verschillende layers te werken en gebruikt te maken van de Graph Shader. Als eerst hadden wij een systeem gemaakt waarin je object N (normal world) in layer A zet en object S (spirit world) in layer B zet. Wanneer de player switch naar de spirit world, verdwijnen alle objecten in layer A en worden alle objecten in layer B getoond.

Dit was voor een lange tijd voldoende, echter was het meestal niet duidelijk dat een object werd geswitched. Het had geen animatie en was daardoor moeilijk te spotten. Als uiteindelijke oplossingen hebben wij een shader gemaakt die laat zien in welke staat het object in zit. Daarnaast zet hij de collision van het object aan/uit.

GhostStruct.cs

Beschrijving

Wanneer GhostStruct.cs input krijgt van InputManager.cs zet hij de bool, ***recording*** en ***playing*** naar true. Wanneer de bool ***recording*** true is, slaat hij elke frame de locatie en rotatie van de speler op in een List.

```
private void Update()
{
    if (recording)
    {
        var newGhostTransform = new GhostTransform(playerTransform);
        recordedGhostTransform.Add(newGhostTransform);
    }
```

Deze List bestaat uit de volgende Struct:

```
public struct GhostTransform
{
    public Vector3 position;
    public Quaternion rotation;

    public GhostTransform(Transform transform)
    {
        position = transform.position;
        rotation = transform.rotation;
    }
}
```

Wanneer de bool, ***recording,*** true is, start hij de ***IEnumerator StartGhost().*** Deze courtine roept dan vervolgens de ***DoWorldEffects()*** aan en zet na X aantal seconden *(ghostDelaySpawn)* de ***IEnumerator MoveGhost()*** aan.

Aan het begin was de ***MoveGhost()*** nog van nut om de ghost te laten bewegen. Echter is dit later vervangen door GhostTrail.cs

Met ***StopGhost()*** wordt ***RevertDoWorldEffects()*** aangeroepen en verwijdert hij alle data uit de ***List <recordedGhostTransform>***.

In ***AddCheckpoint()*** wordt er een checkpoint aangemaakt wanneer de player dit aanroept via de InputManager.cs. Meer informatie hierover in het script zelf.

Theorie & Ontwerpkeuzes

Zoals al eerder is aangegeven, is dit script later in het project aangepast. Omdat we niet makkelijk de player Mesh konden kopiëren via dit script, is het gesplitst in GhostTrail.cs. Daarnaast hadden we het eerst gemaakt dat na X aantal seconden de player automatisch terug werd geplaatst in de normal world. Dit moest anders omdat de timer pas af moet gaan als de player beweegt. Dit is gesplitst naar CalculateStamina.cs De uiteindelijke theorie is dus de player's locatie wordt opgeslagen in een List, wat bestaat uit een Struct en wordt gespeeld door een courtine.

GhostSettings.cs

Beschrijving

Wanneer de ***WorldSwitchManager.OnWorldSwitchEvent()*** geen ***WorldType.Normal***, aangeeft, instantiate hij in de ***FixedUpdate()*** om de X seconden *(ghostSpawnCooldownStarted)* een copy van de player's mesh.

```
private void FixedUpdate()
{
    ghostSpawnCooldown += Time.deltaTime;

    if (spiritWorld && SceneContext.Instance.hasMoved &&
ghostSpawnCooldownStarted &&
        (ghostSpawnCooldown > lastGhostSpawnCooldown))
    {
        var instance = Instantiate(copy, transformC.position,
            transformC.rotation, spawn);

        ghostSpawnCooldown = 0;
    }
}
```

Deze gameObject bestaat uit het volgende:

Op dit object zit het script GhostTrail.cs.

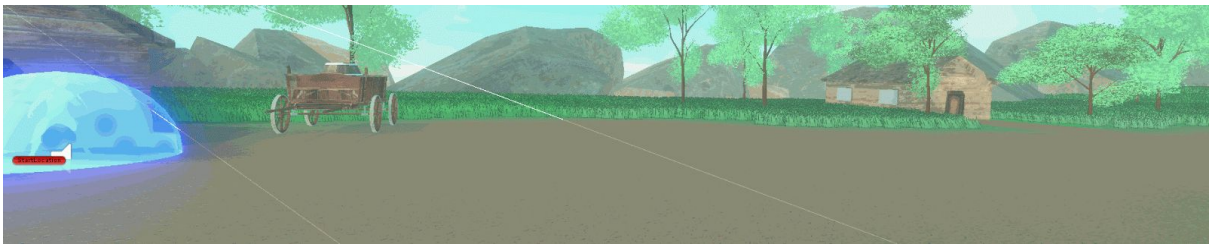
Theorie & Ontwerpkeuzes

Zoals al eerder is aangegeven, is dit script later toegevoegd omdat de mesh van de player kopiëren, moeilijker bleek dan gedacht.

GhostTrail.cs

Beschrijving

Wanneer het GameObject wordt ge-instantiate waarin dit script zich op bevindt, zet hij de material property, "_FadeGhost" naar 0.1. Na dit gedaan te hebben zet hij de ***IEnumerator DeleteGhostSeq()*** aan. Hierin wordt na 1 seconden de "_FadeGhost" naar 1 gezet en wordt het object verwijdert. HIerdoor krijg je dit effect:



Theorie & Ontwerpkeuzes

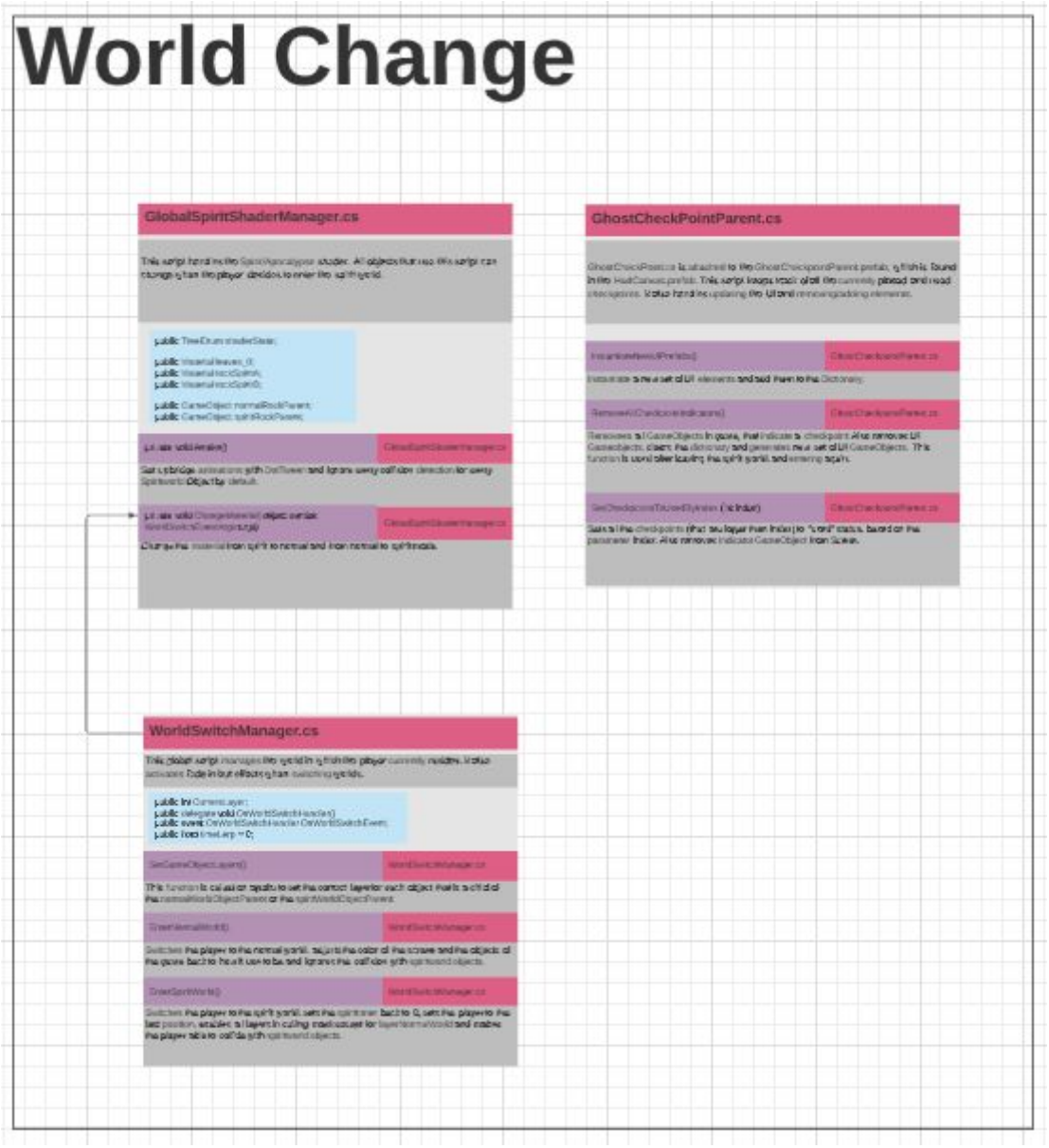
Dit script is is alleen geplaatst op GameObject(GhostCopy) omdat hij zich zelf verwijdert na een aantal seconden.

Activiteit - World Change

Welke scripts spelen een rol in deze activiteit?
GlobalSpiritShaderManager.cs - GhostCheckPointParent.cs - WorldSwitchManager.cs

Wat voor gelinkte scripts zijn er?
PlayerArrowBase.cs

Referentie naar flowchart



Korte beschrijving van de activiteit

GlobalSpiritShaderManager.cs, GhostCheckPointParent.cs en WorldSwitchManager.cs behoren alle drie tot de “World Change” activiteit. Van deze drie zijn de GlobalSpiritShaderManager en de WorldSwitchManager met elkaar verbonden. Namelijk maakt de functie ChangeMaterial() gebruik van de WorldSwitchEventArgumenten en gebruikt die dat als parameter. Verder is the WorldSwitchManager.cs ook nog verbonden met de PlayerArrowBase.cs script, dit omdat hij in de EnterSpiritWorld() functie gebruik maakt van de PlayerArrowBase array.

Theorie

Onderzoeksvraag

Aan het begin van het project, direct na de brainstorm sessies, hadden we al besloten dat een WorldSwitch functie sowieso een van de belangrijkste eigenschappen van onze game zou gaan worden. Eerst beschreven we deze staat van switchen als de “Low Sanity” wereld, een wereld waar je alleen kan komen als je Sanity in het spel heel laag is en eruit ziet als een wereld waar alles anders is.

Uiteindelijk besloten de designers hier een beetje van af te stappen en noemden we deze wereld “The SpiritWorld”, aangezien het witte poppetje die onze karakter tijdens het prototype maken volgde leek op een spirit. We besloten de term “SpiritWorld” erin te houden en besloten de andere wereld niet een low sanity wereld te maken maar juist een soort apocalypse wereld, een wereld in de toekomst waarbij als je die betreed de tijd in de “echte” of “normale wereld, wat langzamer verloopt, waardoor je in de SpiritWorld meer kan bereiken.

Gevonden theorie, Resultaten & conclusie

We besloten dat de beste manier om een “Switch” in onze game te verwerken is door te gaan spelen met de Layers van de game. Zo vroegen we aan onze Artists of ze van alle objecten twee soorten konden maken: een normaal model en een zelfde model die apocalypse themed is. Beiden zetten we neer op precies dezelfde positie en stoppen we in een gameobject, zoals je in de afbeelding hiernaast kan zien.

De normale modellen zetten we dan op de “normal” layer (layer: PersonA) en de Apocalypse modellen zetten we dan op de SpiritLayer (Layer: PersonB). Door Middel van code zorgen we ervoor dat de speler kan switchen tussen de twee layers. Hierdoor ziet de speler de ene keer het ene model en de andere keer het andere.

GlobalSpiritShaderManager.cs

Beschrijving

Dit script handled the Spirit/Apocalypse Shader. Alle objecten die gebruik maken van dit script kunnen switchen naar een andere staat (de Apocalypse staat of de Normale staat) en zodoende visueel de Spirit World betreden.

Theorie & Ontwerpkeuzes

In de Awake() functie zetten we door middel van DotTween de SpiritShader aan. Ook besluiten we in deze functie dat alle vormen van Collision by default uit staat. Dit zorgt ervoor dat je bijvoorbeeld een brug wel kan zien maar er niet overheen heen kan lopen zonder er door heen te zakken.

```
//Ignore collision detection for every spirit object
foreach (var col in spiritColArray)
{
    Physics.IgnoreCollision(playerCol, col, true);
}

//Don't ignore collision detection for every normal object
foreach (var col in normalColArray)
{
    Physics.IgnoreCollision(playerCol, col, false);
}
```

```
//Normal State
if (args.NextWorld == WorldSwitchEventArgs.WorldType.Normal)
{
    leaves_0.DOFloat(0.5f, "_Alpha", 2);
    rockSpiritA.DOFloat(1, "_fadeRock", 1);
    rockSpiritB.DOFloat(0, "_fadeRock", 1);
    foreach (var col in spiritColArray)
    {
        Physics.IgnoreCollision(playerCol, col, true);
    }

    foreach (var col in normalColArray)
    {
        Physics.IgnoreCollision(playerCol, col, false);
    }
}
```

In de ChangeMaterial() functie maken we gebruik van de WorldSwitchEventArgumenten. De WorldSwitchManager is namelijk degene die beslist of de player heeft besloten over te stappen naar de andere wereld. De ChangeMaterial() functie word dus alleen opgeroepen als de de speler is geswitched van wereld.
In deze functie checken we of de switch heeft plaatsgevonden en zetten dan alle “speciale” objecten in onze game om naar objecten van de andere wereld. We gebruiken hiervoor DotTween en de Spiritshader. Een voorbeeld van hoe we dit coderen kan je hiernaast zien.

SHOW SPIRIT SHADER SWITCH

GhostCheckPointParent.cs

Beschrijving

GhostCheckPointParent.cs zit vast aan de GhostCheckPointParent.prefab, die je kan vinden in de HudCanvas.prefab. Dit script houd alle neergeplaatste en gebruikte checkpoints in het spel bij en zorgt ervoor dat wanneer je sterft je niet helemaal opnieuw weer moet beginnen. Ook update dit script de UI en haalt die elementen weg en voegt die ook indien nodig toe.

Theorie & Ontwerpkeuzes

De GhostCheckPointParent.cs maakt gebruik van een Dictionary. Deze Dictionary houd bij welke checkpoint UI GameObjects er in het spel hebben plaatsgevonden en welke scripts ervoor gebruikt zijn. Dit houd die bij doormiddel van Strings die gebaseerd zijn op de hierarchy namen.

```
// Dictionary that keeps track of the checkpoint UI GameObjects and scripts (by string, based on hierarchy names + index i).
private Dictionary<string, GameObject> checkpointUIObjects = new Dictionary<string, GameObject>();
```

De RemoveAllCheckpointIndicators() functie verwijdert alle Checkpoint gameobjecten uit de game. Ook verwijderd hij alle UI GameObjecten, verschoond die de Dictionary en genereerd die een set aan nieuwe UI gameobjecten. Deze functie word gebruikt na het verlaten van de Spirit World en wanneer je er weer terug in komt.

```
for (int i = 0; i < ghostAndWorldSettings.maxCheckpoints; i++)
{
    checkpointUIObjects.TryGetValue("checkpoint_" + i, out GameObject tempGameObject);
    GhostCheckpointUI script = tempGameObject.GetComponent<GhostCheckpointUI>();
    Destroy(script.GetCheckpointIndicator());
    Destroy(tempGameObject);
}
checkpointUIObjects.Clear();
InstantiateNewUIPrefabs();
```

Bij de functie SetCheckPointsToUsedByIndex() gebruiken we de parameter “index”. Index gebruiken we om in onze code aan te geven of een checkpoint al gebruikt is of niet. Als een checkpoint in de hierarchy lager staat dan de gegeven index cijfer, dan is die checkpoint al gebruikt en kan die verwijderd worden.

WorldSwitchManager.cs

Beschrijving

Dit globale script manages de wereld waarin onze speler momenteel voor kiest om in te leven. Dit kan de normale wereld zijn of de spirit wereld. Ook activeer je door middel van dit script de fade in/out effecten die je ziet tijdens het switchen van werelden.

Theorie & Ontwerpkeuzes

In dit script zijn er een aantal belangerijke functies die en prive en public staan. Een voorbeeld van een belangerijke functie is de “SetGameObjectLayers()” functie die bepaald welke Gameobjecten allemaal gaan switchen wanneer je van wereld switched.

Verder hebben we nog twee public functies die best wel veel op elkaar lijken qua opzet, de “EnterNormalWorld()” functie en de “EnterSpiritWorld” functie. Beide zijn functies zetten de wereld waar onze speler zich bevind op. De functies vernieuwen of starten de SpiritTimer, verplaatst de speler van positie en geeft aan op welke Layer de player nu aan het kijken is. Ook geeft die aan of de objecten van die layer nog steeds een collision hebben of niet. De switch word hierin gedaan doormiddel van Dottween die een ingebouwde fade in/out functie ervoor gebruikt.

Een voorbeeld van hoe de script van “EnterSpiritWorld” eruit ziet kan je hieronder vinden:

```
//set spirittimer to 0;
calculateStamina.spiritTimer = 0.0f;
worldIsNormal = false;

//Set playerTransform to last position
var playerPos = playerObject.transform.position;
DOTween.Complete(materialFade);

// Enables all layers in culling mask except for layerNormalWorld
mainCamera.cullingMask = ~(1 << LayerNormalWorld);

//playerSpiritWorldEffect.SetActive(true);
Physics.IgnoreLayerCollision(PlayerLayer, LayerSpiritWorld, false);
Physics.IgnoreLayerCollision(PlayerLayer, LayerNormalWorld, true);
Instantiate(shockwavePrefab, playerPos, Quaternion.identity, playerObject.transform);
Instantiate(spiritWorldParticleEffect, playerPos, Quaternion.identity, playerObject.transform);
```

Beide public functies zijn verbonden aan de PlayerArrowBase.cs script. Dit omdat ze de script oproepen en ervoor zorgen dat eenmaal geswitched naar een andere wereld je je eigen arrows nog steeds kan gebruiken.

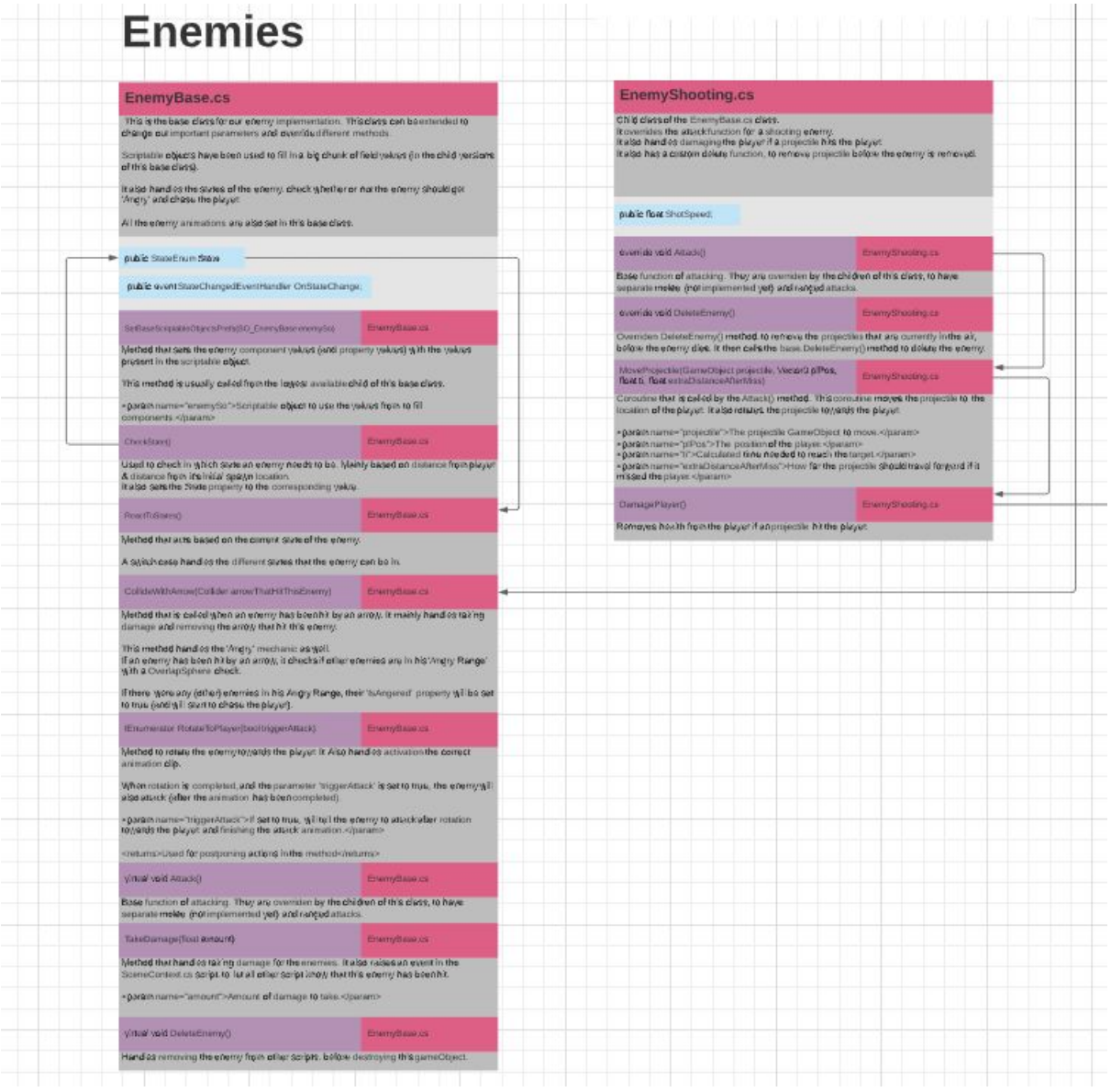
```
//Arrows
PlayerArrowBase[] arrows = GameObject.FindObjectsOfType<PlayerArrowBase>();
foreach (var arrow in arrows)
{
    arrow.SwitchArrowsToSpiritWorld();
}
```

Activiteit - Enemies

Welke scripts spelen een rol in deze activiteit?
EnemyBase.cs - EnemyShooting.cs - EnemyShootingFast.cs

Wat voor gelinkte scripts zijn er?
SettingsEditor.cs - BowAndArrowSettingsModel - EnemySettingsModel - PlayerController.cs - NavMeshAgent

referentie naar flowchart



Het volledige diagram is te zien op:
https://lucid.app/lucidchart/78372da9-b978-47aa-9434-fcf67e36c00a/view#?folder_id=home&browser=icon

Korte beschrijving van de activiteit

De enemies in onze game maken gebruik van een state machine. De states worden elke update tick gecheckt (voornamelijk afhankelijk van de afstand van de enemy ten opzichte van een ander punt) en elke update wordt de corresponderende actie toegepast. Elke enemy kent in principe 4 verschillende instelbare bereiken.

1. De territorial range, die wordt gebruikt om de afstand tussen de enemy en zijn spawnpoint te berekenen (dus waar hij in de editor is neergezet)
2. De sight range, deze wordt gebruikt om te weten wanneer hij de speler ‘ziet’ en dus op de speler af begint te lopen
3. De attack range, deze wordt gebruikt om te stoppen met lopen en de speler aan te vallen
4. De Angry range, deze wordt gebruikt om andere enemies op de state ‘Angry’ te zetten wanneer een enemy wordt geraakt.

De enemies schieten elkaar dus te hulp wanneer er één van de enemies geraakt wordt.

Theorie

Onderzoeksvraag

Wat is de beste manier om enemies toe te voegen aan ons spel, die redelijk zelf nadenken en ook niet te resource intensief zijn?

Gevonden theorie & Resultaten & conclusie

Er zijn een aantal methodes om enemies te programmeren. Snel is gevonden dat Unity een heel mooi systeem heeft om de Pathfinding van de enemies te regelen, dat gelijk werkt op Unity Terrain Components (wat er gebruikt wordt in ons spel). Deze zal dus sowieso benut worden, aangezien we dan zelf geen pathfinding hoeven te maken, of nog een third party library hoeven te downloaden. Echter zijn er wel meerdere manieren of de states van

enemies te bepalen. Bijvoorbeeld door een neural network (echte zelflerende AI), wat helaas al snel buiten de scope van dit projectduur zou vallen. Of door verschillende ‘States’ te maken waarin de enemies kunnen zijn. Hier hebben we uiteindelijk voor gekozen. Hoe dit precies werkt staat uitgelegd in de beschrijving van de drie gebruikte scripts.

EnemyBase.cs

Beschrijving

De EnemyBase.cs class is het main script van het enemy systeem. Deze class heeft alle methods en properties die gelden voor alle soort types enemies die we zouden willen maken. Denk aan, de state machine, het draaien richting de speler en een base Attack method die overschreven kan worden door de children van deze classes.

Om alle benodigde waardes te vullen die de enemies nodig hebben (attack speed, attack cooldown, movement speed, angular speed etc) worden van hun waardes voorzien door middel van scriptable objects. Deze scriptable objects vullen de variabelen in de child classes gebaseerd op de enemy base class. Net als de Attack() method, deze is leeg in de EnemyBase.cs class, maar wordt gevuld in zijn children.

De state machine kent de volgende states:

```
public enum StateEnum
        {
            Idle = 0,
            Chase = 1,
            Attack = 2,
            ReturningHome = 3,
            Dead = 4
        }
```

Zoals eerder vernoemd checkt de enemy naar welke state hij moet aan de hand een paar verschillende afstanden, dit wordt bepaald met de volgende method:

```
protected virtual void CheckState()
        {
            switch (DistanceToPlayer)
            {
                case var ex when ex > SightRange && !IsAngered:
                    State = StateEnum.Idle;
                    break;

//When distance is smaller than AttackRange && distance is smaller than
attackRange
                case var ex when (ex < AttackRange && ex < SightRange)
||
                                    (State == StateEnum.Chase && IsAngered
&& (ex < AttackRange && ex < SightRange)):
```

```
                    State = StateEnum.Attack;
                    break;

                case var ex when (ex < SightRange) || (IsAngered && ex >
SightRange):
                    State = StateEnum.Chase;
                    break;
            }

            if (State == StateEnum.Chase || State == StateEnum.Attack)
            {
                //Debug.Log("CurrentlyChasing or attacking");
            }
            else if ((DistancePlayerToInitPoint > TerritorialRange &&
BackHomeDistance >= 1f) && !IsAngered)
            {
                //Debug.Log("Am not chasing or attacking, and need to
return home");
                State = StateEnum.ReturningHome;
            }
        }
```

Wanneer de enemy op de correcte state wordt gezet, kan daarna (in dezelfde update cycle) gereageerd worden op deze state (code van de method wordt buiten dit verslag gelaten voor overzichtelijkheid, zie: EnemyBase.cs -> ReactToStates();)

Theorie & Ontwerpkeuzes

Om te zorgen dat de enemies toevoegbaar blijven, is de Base class geschreven, zodat er allemaal verschillende type enemies gemaakt konden worden, zonder dat er dubbele code met dezelfde functie geschreven hoefde te worden. De Attack() method is bijvoorbeeld niet ingevuld in de base class, en moet nog ingevuld worden in de child.

EnemyShooting.cs

Beschrijving

Dit is een child van de EnemyBase.cs class. In deze class wordt de Attack functie overschreven, en een kleine aanpassing aan de death function gemaakt (die alle geschoten projectiles van de enemy verwijderd, voordat de enemy wordt verwijderd).

```
protected override void DeleteEnemy()
    {
        foreach (var obj in spawnedProjectiles)
        {
            var script = obj.GetComponent<EnemyShootingProjectile>();
            script.SetEnemyThatShotProjectile(null);
            Destroy(obj);
        }

        base.DeleteEnemy();
    }
```

De attack functie is om een projectiel naar de speler toe te bewegen. Wanneer dit projectiel de speler raakt, zal er levens van de speler zijn health afgehaald worden.

Theorie & Ontwerpkeuzes

Het idee achter deze class was in eerste instantie om nog meer children van te kunnen maken. Bijvoorbeeld een snelle ranged enemy, en een slome range enemy. De snelle ranged enemy zou dan een ‘bult’ of iets dergelijks op zijn rug krijgen, zodat je deze alleen kon verslaan door je spirit mode te gebruiken. Echter is dit door tijdgebrek er nooit meer van gekomen, maar bestaat de child versie van deze class nog wel (EnemyShootingFast.cs).

Verder is ervoor gekozen om de projectiles niet met rigidbodies en force te bewegen, maar om met onze geliefde library DoTween de transform over de tijd te verplaatsen. Aangezien DoTween een tijdsduur nodig heeft om te weten hoelang het moet duren voordat het projectiel bij zijn eindbestemming is, en aangezien we de snelheid van het projectiel zelf willen kunnen bepalen, is er een kleine omrekening gemaakt.

De afstand tussen het projectiel en de speler wordt bepaald, en dan wordt de formule van afstand (s = v * t, waar s: afstand [m], v: snelheid [m/s], t: tijd [s]) omgedraaid om zo de benodigde tijd te berekenen:

```
var dist = Vector3.Distance(playPos, bulletSpawn.transform.position);
var time = dist / ShotSpeed;
```

```
protected override void DeleteEnemy()
    {
        foreach (var obj in spawnedProjectiles)
        {
            var script = obj.GetComponent<EnemyShootingProjectile>();
```

```
            script.SetEnemyThatShotProjectile(null);
            Destroy(obj);
        }

        base.DeleteEnemy();
    }
```

EnemyShootingFast.cs

Beschrijving

Dit is de (simpele) child class van EnemyShooting. Deze wordt momenteel alleen gebruikt om variabelen te vullen:

```
public override void Awake()
    {
        base.Awake();
        SetBaseScriptableObjectsPrefs(enemyBaseSo);
        SetChildScriptableObjectsPrefs();
        DrawGizmos = true;
    }

    private void SetChildScriptableObjectsPrefs()
    {
        ShotSpeed = enemyBaseSo.shotSpeed;
    }
```

Theorie & Ontwerpkeuzes

Het idee achter deze class was in eerste instantie om bijvoorbeeld een snelle ranged enemy, en een slome range enemy te maken. De snelle ranged enemy zou dan een ‘bult’ of iets dergelijks op zijn rug krijgen, zodat je deze alleen kon verslaan door je spirit mode te gebruiken (want je beweegt dan zelf ook sneller). Echter is dit door tijdgebrek er nooit meer van gekomen, maar bestaat deze class dus nog wel. De waardes van de scriptable objects kunnen uiteraard ook veranderd worden om gevoelsmatig toch andere enemies te creëren.

Activiteit - Interaction

Welke scripts spelen een rol in deze activiteit?
IInteractable.cs, Interactable.cs

Wat voor gelinkte scripts zijn er?
PlayerController.cs

referentie naar flowchart
Zie Interaction gedeelte

Korte beschrijving van de activiteit

Binnen de game moet de player met verschillende soorten objecten kunnen interacten op een standaard manier. Een interactable is een GameObject dat van alles kan zijn. Elk interactable kan ook zijn eigen unieke gedrag hebben wanneer de player probeert te interacten.

De process zelf is vrij simpel:

1. Player kijkt naar een interactable object.
2. De interactable zal een highlight effect krijgen
3. De speler zal de optie krijgen om op een knop te drukken
4. De object zal een gedrag tonen wanneer de speler interact.

Theorie

Onderzoeksvraag

De balangrijkste vraag was hoe we een systeem kunnen maken waarbij je objecten interactable kan maken zodat de speler er na toe kan lopen en de optie kan krijgen om interactie te doen. Het belangrijkste is dat elk interactable zijn eigen unieke gedrag kan tonen bij interactie.

Gevonden theorie & Resultaten & conclusie

Om een systeem te maken voor deze interactie moest er gebruikt gemaakt worden van abstractie met behulp van interfaces en base classes.

Alle interactable moeten hun eigen code kunnen uitvoeren als de player interactie met ze uitvoert. Er kan dus geen generieke class gemaakt worden die op alle interactable kan worden toegepast. Ook moeten we zo min mogelijk dubbel code schrijven.

Als we kijken naar de interactables, dan zien we dat ze allemaal de volgende eigenschappen delen:

- Ze kunnen allemaal gebruikt worden als de speler dichtbij komt, naar de object kijkt en op E drukt.
- Ze moeten allemaal dezelfde highlight effect hebben als speler naar ze kijkt.

- Ze tonen allemaal een text prompt in de vorm van 'Press [E] to {interaction werkwoord}

Het enige verschil tussen de interactables is puur wat ze doen op moment van interactie.

In zulke situaties komen C# Interfaces heel handig. Ik heb eerst een interface genaamd **IInteractable** gemaakt die alle benodigde functionaliteit heeft:

```
public interface IInteractable
{
    string InteractionVerb { get; }

    GameObject GameObject { get; }

    bool IsLocked { get; set; }

    void OnPlayerInteract();

    void SetHighlight(bool enable);
}
```

Elke object dat een interactable wilt zijn moet dan een script hebben die deze interface implementeert. In de Player Manager word er vervolgens elke frame gekeken of de speler naar een interactable kijkt, zoja dan kunnen we de highlight effect aan zetten via **SetHighlight** en de interaction text op het scherm tonen met behulp van **InteractionVerb**. Als de speler dan vervolgens op de interactie knop drukt, zal de **OnPlayerInteract** method opgeroepen worden.

Dankzij deze methode kan de PlayerManager dus makkelijk herkennen of iets een interactable is en daarmee ook interacteren zonder dat het de daadwerkelijke implementatie van de script hoeft te weten. Dit zal ervoor zorgen dat je onbeperkt type interactables kunt hebben.

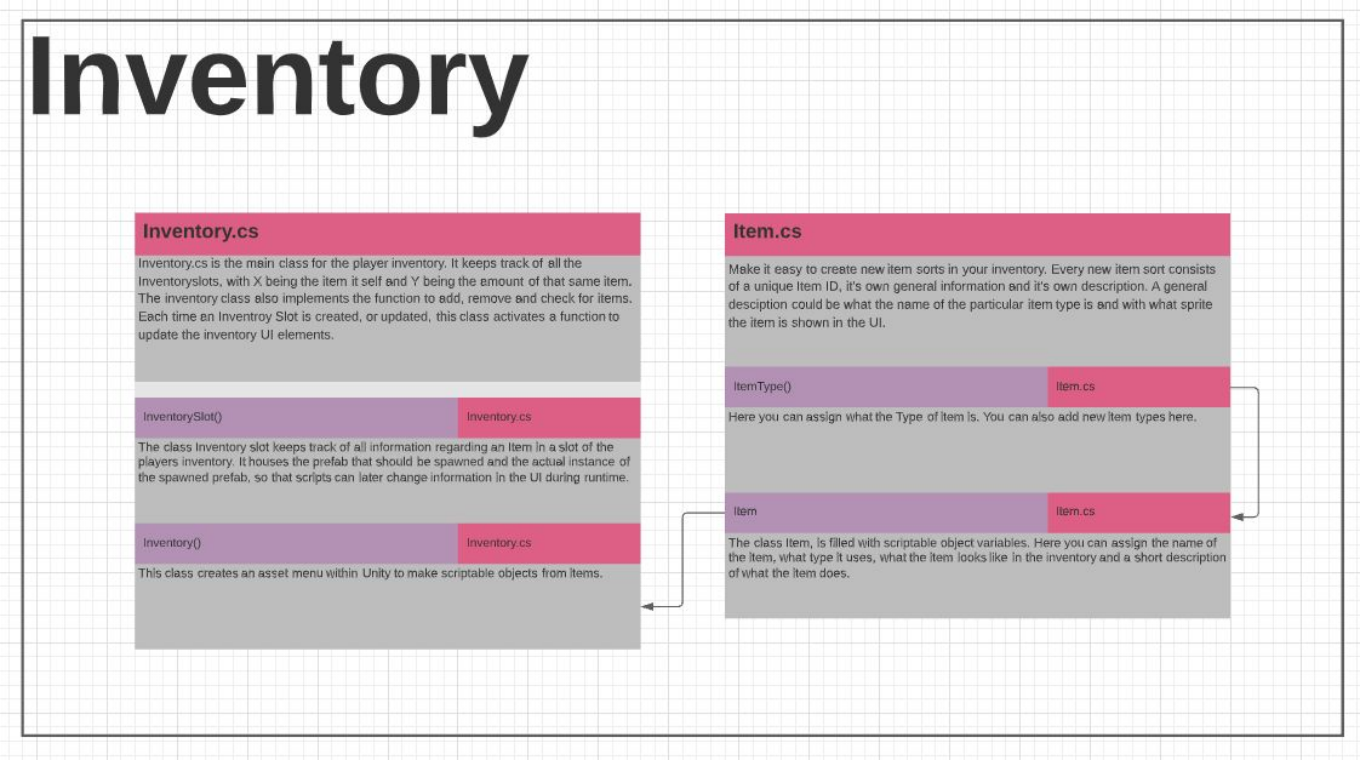
Verder is ook de abstracte class **Interactable** gemaakt die de interface implementeert. Het doel van deze class is om het maken van interactables simpel te maken door alle gedeelde basis functionaliteit hierin te zetten.

Activiteit - Inventory

Welke scripts spelen een rol in deze activiteit?
Inventory.cs - Item.cs

Wat voor gelinkte scripts zijn er?
TestKeyInteractable.cs - TestKeyRemoveInteractable.cs - ItemKey.cs - ItemQuestObject.cs

Referentie naar flowchart



Korte beschrijving van de activiteit

Inventory.cs & Item.cs zijn de 2 belangrijkste scripts die de Inventory system vormen. Inventory.cs houd bij alle inventory items bij en Item.cs is een kleine script die vastgelegd wat de *type* van de items zijn die je kan oppakken.

Theorie

Onderzoeksvraag

Origineel bestond het inventory systeem uit meer scripts, dit hebben we verminderd. De reden hierachter is dat origineel we een inventory system wouden voor onze puzzle spellen. Het idee was dat we verschillende soorten puzzles zouden hebben waarbij het belangrijk is om van objecten van bijvoorbeeld de spiritworld op te pakken en te gebruiken in de normale wereld.

Helaas had dit project nogal wat vertraging. We onderschatten hoe ingewikkeld het zou zijn om een inventory systeem in elkaar te zetten. Uiteindelijk was het ons grotendeels gelukt om een inventory systeem op te zetten, maar had de inventory systeem nog steeds een aantal bugs. Zo kon je in de UI bijvoorbeeld een 3D item niet makkelijk ronddraaien in je canvas.

Uiteindelijk besloten we de UI van het inventory systeem er grotendeels uit te halen. Het was te veel moeite en aangezien het project toch al zoveel vertraging had, besloten de designers in de tussentijd om puzzles te bedenken die amper tot geen inventory systeem nodig hebben.

Nu gebruiken we inventory systeem alleen nog maar voor een puzzle in level 1 en heb je voor die puzzle geen inventory UI nodig om te accessen.

Inventory.cs

Beschrijving

Inventory.cs is de basis class for the inventory van de player. Het houd bij welke Inventory Slots gevuld zijn en hoeveel van hetzelfde item een speler heeft opgepakt. Inventory.cs maakt gebruik van functies die items toevoegt, weghaalt en checkt voor de inventory. Elke keer wanneer een inventory slot is gecreëerd of geupdate wordt de class geactiveerd die de UI elementen van de inventory update (ook al wordt de UI niet meer gebruikt in het spel). Inventory.cs bestaat uit twee classes: InventorySlot en Inventory.

Theorie & Ontwerpkeuzes

De class “InventorySlot” houd alle informatie over een item in een slot van de players inventory. Het bewaard de prefab dat zou moeten verschijnen in de UI en behoud de instance daarvan. Hierdoor kan het script later de informatie van de UI (als je bijvoorbeeld een nieuw of ander item oppakt) tijdens run time updaten.

Deze class geeft een item een ID nummer, een bijhorende prefab en houd bij of het aantal van het item dat is opgepakt of weggegooid meer of minder moet worden volgens de acties van de speler.

De class “Inventory” zorgt ervoor dat het toevoegen van een Item binnen in Unity makkelijk gaat. Het creeerd namelijk een Asset Menu in unity waarbij je met een druk op de knop makkelijk scriptable objects kan maken van items.

```
[CreateAssetMenu(fileName = "NewMainInventoryObject", menuName = "Inventory/InventoryMain")]
```

In deze class word er een dictionary aangemaakt die het aantal slots bij houd. Door de AddItemToInventory() functie kan je een item toevoegen. Als die inventory slot nog niet bestaat dan creeerd deze functie er zelf eentje.

```
foreach (var slot in slots)
{
    if (slot.Key == item.uniqueItemId)
    {
        //Debug.Log("Player already had a slot with this item, adding the amount now.");
        slot.Value.AddAmount(amount);
        hasItem = true;
        break;
    }
}

if (!hasItem)
{
    //Debug.Log("Player does not have item already, make new Inventory Slot");
    slots.Add(item.uniqueItemId,
        new InventorySlot(item.uniqueItemId, item, amount,
            item.uiElementPrefab));

    //Convert the current dictionary values to a list, so that the json formatter can save and read it
    inventoryList = slots.Values.ToList();
}
```

Verder maakt deze class ook nog gebruik van een “RemoveItemFromInventory” functie die bijhoud of het aantal items onder de nul zit of niet. De functie “GetInventorySlotByItem” maakt gebruik van de InventorySlot class die ook in deze script zit en wordt gebruikt om de gevraagde item van de geklikte inventoryslot te krijgen.

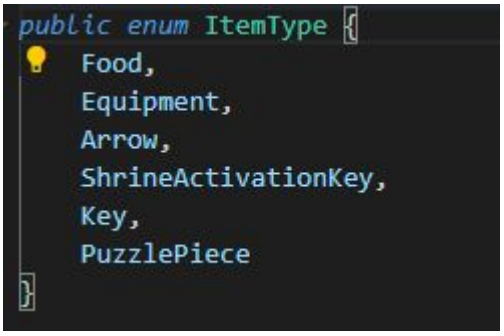
Item.cs

Beschrijving

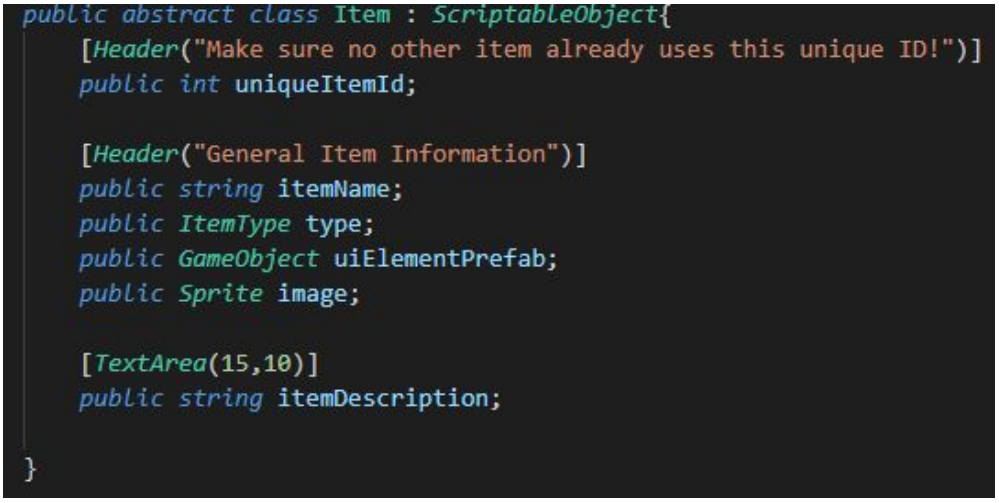
Item.cs maakt het makkelijk om een nieuwe item type aan te maken voor in de inventory. Elke nieuwe Item krijgt een eigen unieke ID, generale informatie zoals een naam, image en eigen UIPrefab en een toegevoegde ItemDescriptie. Item.cs word gebruikt door de class Inventory en maakt scriptable item objects aan.

Theorie & Ontwerpkeuzes

Bij de functie ItemType() worden er eigenlijk alleen maar de soorten Items waaruit je bij het maken van een Item kan kiezen aangemaakt.



Bij de Item Class kan je dan wanneer je een Item hebt aangemaakt de ItemType eraan toevoegen. Ook kan je in deze class via Unity zelf bij een nieuwe item een prefab eraan toevoegen, een image en bijhorende descriptie.



IInteractable

Beschrijving

Dit is de interface van het interactie systeem. Alle interactable objects moeten een script bevatten die deze Interface implementeert. Hierdoor kan de PlayerController de interactable objects herkennen en gebruiken.

Theorie & Ontwerpkeuzes

Door gebruik te maken van interfaces is het mogelijk gemaakt voor de PlayerController om interacties te herkennen en te gebruiken zonder dat het de onderliggende implementatie hoeft te weten. Hierdoor kan je zoveel interactable scripts maken als je wilt. Zolang ze allemaal de basis interface implementeren zal de PlayerManager het verschil niet kunnen zien.

Hierdoor kan elk script zijn eigen implementatie hebben voor de **OnPlayerInteract** method.

Interactable

Beschrijving

Om het maken van interactable objects nog makkelijker te maken was er ook een abstract class gemaakt die de interface implementeert. Het is aanbevolen om direct vanaf deze class over te nemen in je child classes.

Theorie & Ontwerpkeuzes

Origineel was het puur met alleen de interface, maar tijdens verdere ontwikkeling begon er heel veel dubbel code te ontstaan in elk script die **IInteractable** implementeert. De code die de highlight materiaal aanmaakt en aan/uit zet was identiek op elke script, daarom was de **Interactable** class geïntroduceerd zodat het alle basisfunctionaliteit kan bevatten voor alle scripts. Hierdoor is er geen dubbel code wat mooi overeenkomt met de **DRY** principe (Dont-Repeat-Yourself).

Activiteit - Player sounds during animations

Welke scripts spelen een rol in deze activiteit?
PlayerSoundStateBehavior.cs

Wat voor gelinkte scripts zijn er?
N/A

referentie naar flowchart
Zie *PLAYER ANIMATION SOUND* gedeelte

Korte beschrijving van de activiteit

Voor het afspelen van bepaalde player SFX is het belangrijk dat het op bepaalde momenten gebeurd tijdens een animatie. De boog geluiden moeten bijvoorbeeld afgespeeld worden zodra de speler de boog schiet animatie doet. De script zelf is redelijk simpel, maar omdat deze methode ook voor enemies gebruikt wordt is het handig om dit als voorbeeld te zetten hier.

Theorie

Onderzoeksvraag

De vraag was hoe we geluiden kunnen afspelen wanneer de Animator van een Player bepaalde animatie states binnen gaat of weggaat.

Gevonden theorie & Resultaten & conclusie

Binnen elk animatie state kan je behaviors toevoegen. Je kan in Unity ook je eigen scripts maken die als behavior kunnen worden toegevoegd. Om dit te doen moet je een class maken die de Unity **StateMachineBehavior** class overneemt. Je kan de base methods **OnStateEnter** en **OnStateExit** overriden. Deze methods worden automatisch opgeroepen wanneer je een state binnengaat of weggaat waarop deze script is toegevoegd.

Binnen deze functie worden verschillende parameters meegenomen. Via deze parameters kan je de naam van de state opvragen. Door de naam te checken kan ik kiezen welk geluid ik wil afspelen.

PlayerSoundStateBehavior

Beschrijving

Dit is de class die als behavior is gekoppeld aan een aantal animatie states in de Player Animator. Deze script neemt de **StateMachineBehavior** class over.

Theorie & Ontwerpkeuzes

Het belangrijkste was dat er bepaalde geluiden kan afspelen wanneer je een animatie state verlaat of binnengaat. Dat gaat redelijk makkelijk dankzij de twee methods die naar deze events kijkt.

Activiteit - Level background music

Welke scripts spelen een rol in deze activiteit?
LevelAudioManager.cs

Wat voor gelinkte scripts zijn er?
SceneContext.cs

referentie naar flowchart
Zie *LEVEL AUDIO* gedeelte

Korte beschrijving van de activiteit

Bij elk level is zijn er verschillende achtergrondmuziek die afgespeeld moeten worden afhankelijk van wat er aan de hand is. De volgende geluiden zijn er:

- Normale achtegrond muziek
- Spirit world muziek
- Gevechtsmuziek
- Doodgaan muziek

Theorie

Onderzoeksvraag

Hoe kunnen we een systeem hebben waarbij de verschillende achtergrondmuziekje afgespeeld kunnen worden in de juiste omstandigheden?

Gevonden theorie & Resultaten & conclusie

Eerst moet er gekeken worden hoe de verschillende audio geluiden afgespeeld kunnen worden. In Unity heb je de AudioSource component dat geluid kan afspelen. Hiervoor is er voor elk geluid een gameobject gemaakt die een audiocomponent bevat. Door verschillende audio components te hebben is het ook makkelijker om een fade effect te hebben wanneer we overgaan van de ene muziek naar de andere.

Binnen de LevelAudioManager script moeten we een manier hebben om erachter te komen wanneer bepaalde geluiden moeten worden afgespeeld.
Het beste manier hiervoor is om gebruik te maken van Events. Zowel de enemy en player scripts hebben bepaalde events waar we erachter kunnen komen wanneer de player bijvoorbeeld levens kwijtgeraakt heeft.

Via deze events kunnen we makkelijk dus erachter komen wat afgespeeld moet worden. Dankzij helper functies van DOTween is het makkelijk om de volume van een AudioSource langzaam te faden.

Ook is het belangrijk dat de maximale muziek volume instelling in de settings menu correct gerespecteerd wordt. Dankzij de **OnSettingsSaved** event in de SettingsEditor kunnen we makkelijk erachter komen of de settings verandered zijn en vervolgens de maximale volume opnieuw berekenen.

LevelAudioManager

Beschrijving

Dit is de script die op de LevelAudioManager GameObject zit en dus de controle heeft over welke muziek er momenteel afgespeeld moet worden.

Theorie & Ontwerpkeuzes

Het belangrijkste hier is het makkelijk erachter kunnen komen wanneer bepaalde events gebeuren zoals een player die geraakt wordt. Voor elke achtegrond muziek kijken we naar de volgende events:

- Normale achtegrond muziek
    - Speelt standaard af
    - Begint met spelen wanneer de **OnWorldSwitchEvent** wordt afgevuurd en de speler naar de **Normale** wereld overgaat.
- Spirit world muziek
    - Begint met spelen wanneer de **OnWorldSwitchEvent** wordt afgevuurd en de speler naar de **Spirit** wereld overgaat.
- Gevechtsmuziek
    - Begint wanneer de Enemy **OnEnemyHitEvent** gebeurd. Dit is wanneer de speler een vijand raakt.
    - Begint wanneer de Player **OnHealthChangeEvent** gebeurd. Het zal alleen afspelen als de speler health verloren heeft.
- Game Over (Death) muziek
    - Begint wanneer de Player **OnHealthChangeEvent** gebeurd en de player health nul is.

Het overgaan van de ene muziek naar de andere wordt gedaan met behulp van de **DOFade()** method die een Audio volume langzaam naar nul zal brengen. Voor het geluid dat wel afgespeeld moet worden zal het de volume langzaam naar het maximale brengen. Het maximale volume is afhankelijk van de audio instellingen in de menu.

Voor deze script was de keuze gemaakt om gebruik te maken van Events zodat er zo min mogelijk polling plaatsvindt in de Update loop van deze script. Hierdoor is de code ook makkelijker te lezen.

Activiteit - Puzzle - Paired Orbs

Welke scripts spelen een rol in deze activiteit?
PairedActivator.cs

Wat voor gelinkte scripts zijn er?
N/A

referentie naar flowchart
Zie *PairedActivator.cs* gedeelte onder **Interaction**

Korte beschrijving van de activiteit

Als onderdeel van de Paired Orbs puzzel in Level 3 moet er een systeem zijn waar orbs aan elkaar gekoppeld moeten zijn en beide geactiveerd moeten worden op dezelfde tijd zodat de actief blijven. Als de gekoppelde paar niet beide actief zijn binnen een korte aantal seconden, zullen de orbs weer uitgaan.

Theorie

Onderzoeksvraag

Hoe kunnen Orbs aan elkaar gekoppeld worden en vervolgens geactiveerd worden door beide in de paar te activeren.

Gevonden theorie & Resultaten & conclusie

Elke orb heeft nu meerdere states die het kan hebben. Het kan inactief zijn, actief zijn, of ook actief zijn terwijl het zit te wachten tot de andere paar ook geactiveerd wordt.

Wanneer een orb actief is moet de speler dit ook visueel kunnen zien. Dit kan het best gedaan worden door twee GameObjects te hebben voor elk orb, eentje die de actieve orb is en de andere die de inactieve orb is. De script zal dan switchen tussen deze twee objecten.

Verder moet het een referentie hebben naar de PairedActivator script die op de gepaarde orb zit. Hiermee kan het controleren of de gepaarde orb actief is of niet.

PairedActivator

Beschrijving

Dit is de script die de state van elk orb beheerd.

Theorie & Ontwerpkeuzes

Om de staat van elk orb te kunnen beheren is er een enum gemaakt met elk mogelijke waarde die de orb staat kan hebben:

- NotActive
- AwaitingActivation
- FullyActivated

De orb is ook een interactable zodat de speler dus de orb kan aanzetten. Dit betekent dat de script overneemt van **Interactable** base class.

Wanneer de speler de orb aanzet zal de staat naar **AwaitingActivation** gaan. Deze staat betekent dat de orb actief is, maar nog zit te wachten op de gepaarde orb. Als de gepaarde orb niet actief is binnen x aantal secondes, dan zal de staat terugvallen naar **NotActive**.

Als de gepaarde orb wel actief is, dan zal de staat overgaan naar **FullyActivated** en zal het dus permanent aanblijven.

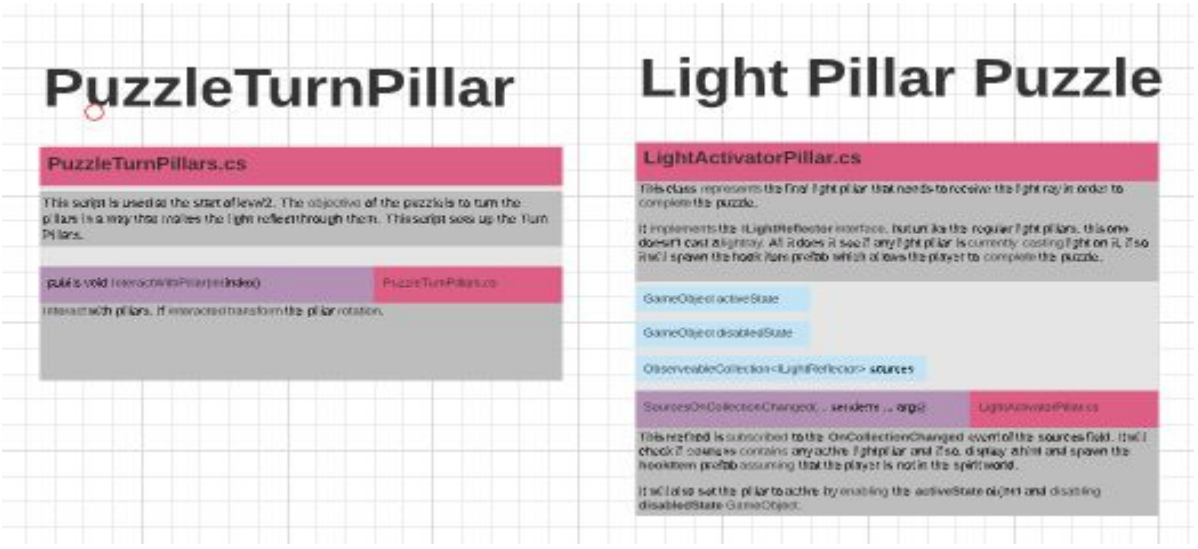
Origineel zou er gebruik gemaakt worden van Events om in de gaten te houden of de gekoppelde orb actief was of niet, maar dit zou de **CheckPairedActivation()** co-routine onnodig complexer maken, daarom word de huidige state van de gekoppelde orb direct gelezen.

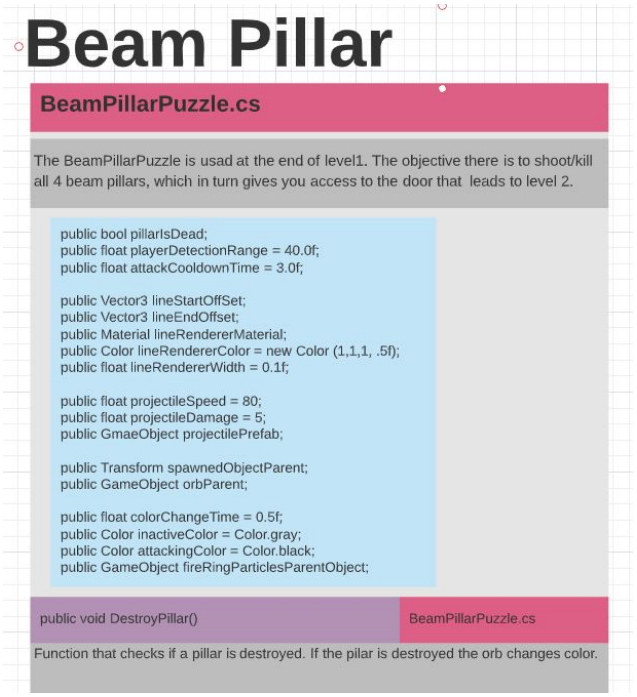
Activiteit - Puzzle - Puzzle Scripts

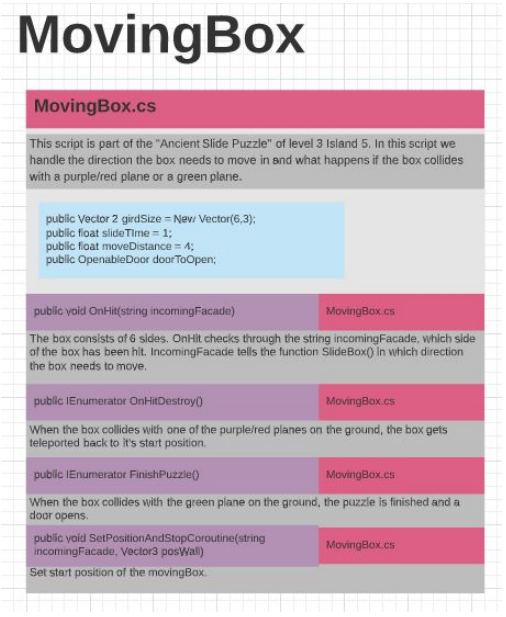
Welke scripts spelen een rol in deze activiteit?
LightActivatorPillar.cs, PuzzleTurnPillar.cs, BeamPillarPuzzle.cs, MovingBox.cs

Wat voor gelinkte scripts zijn er?
RespawnManager & ILightReflector

Referentie naar flowchart







Korte beschrijving van de activiteit

De laatste hoofdstuk van de TO is specifiek toegewijd aan de door ons gebruikte Puzzle scripts. Van deze 4 scripts zijn alleen maar de PuzzleTurnPillar.cs & de LightActivatorPillar.cs scripts met elkaar verbonden. De andere twee staan los van elkaar.

De “PuzzleTurnPillar.cs” en “LightActivatorPillar.cs” zijn scripts die we gebruikt hebben in level 2. Hierbij is het doel door deze torens op een bepaalde manier te draaien moet het licht uiteindelijk de activatie toren raken. Wanneer dit gebeurd kan het puzzel afgemaakt worden. Hiervoor moet dus een toren gemaakt worden die de licht van de lichttorens kan opvangen.

De BeamPillarPuzzle wordt gebruikt aan het einde van level1 en de MovingBox puzzle wordt gebruikt in level 3 eiland 5.

LightActivationPillar.cs

Beschrijving

Voor het ontvangen en uitzenden van licht bestaat er al een Interface (**ILightReflector**) die gebruikt wordt door de licht torens. Omdat de onze activatie toren ook licht moet opvangen is het dus een goed idee om deze interface te implementeren. Het enige verschil is dat wij zelf geen licht uitzenden.

Theorie & Ontwerpkeuzes

De implementatie zal een beetje lijken op de script voor de licht torens. Het zal zelf ook de **ILightReflector** interface gebruiken waardoor het in staat is om licht op te vangen.

Maar in plaats van dat deze toren licht zal weerkaatsen zal het juist niks uitzenden. In plaats daarvan zal het een aantal items spawnen die nodig zijn om de puzzel af te maken.

Het belangrijkste property in dit script is de LightSources list wat bijhoud hoeveel lichtstralen onze toren raken.

In onze script hebben deze list geimplementeerd met als type **ObserveableCollection**. Dit is hetzelfde als een list, maar dan met de mogelijkheid om via events bij te houden wanneer de list verandered wordt.

Omdat **ObserveableCollection** de **IList** interface implementeerd is het geen probleem dat wij deze type gebruiken voor onze LightSources property (wat gedefineerd is als een **IList** in de interface).

Iedere keer wanneer een lichtstraal de toren raakt zal de **SourcesOnCollectionChanged** method aangeroepen worden. Dit zal kijken of er momenteel een lichtstraal de toren aanraakt in de normale wereld, en zal vervolgens de puzzel onderdelen in de level spawnen. Ook zal het een respawn punt aanzetten bij deze gebied omdat de puzzel af is.

PuzzleTurnPillar.cs

Beschrijving

Dit script word gebruikt bij het begin van level2. Het doel van het script is dat de speler de light pillars op zo’n manier kan gaan ronddraaien dat de pilaar licht door reflecteert naar de dichtsbijzijnste andere pilaar. Dit zorgt er op zijn beurt dan voor dat er een soort slang van licht ontstaat.

Theorie & Ontwerpkeuzes

Dit script bestaat uit twee hoofd functies, CheckPillarRotation() die checkt of alle pillars op een correcte wijze zijn rond gedraaid en InteractWithPillar() die checkt of de player op de pillar heeft geklikt of niet.

BeamPillarPuzzle.cs

Beschrijving

The Beam Pillar puzzle word gebruikt aan het einde van level 1. Het doel hierbij is om op de 4 killers te schieten en ze te “killen”. Als je ze alle 4 “kapot” hebt geschoten gaat de deur naar level 2 open.

Theorie & Ontwerpkeuzes

De script maakt gebruik van meerdere hoofd functies. ChangeOrbColor() zorgt ervoor dat de kleur van de orb met DotTween langzaam veranderd en maakt gebruik van de functie “SetScaleOfParticles()”.

```
switch (indexColor)
{
    case 0:
        SetScaleOfParticles(.7f, colorChangeTime);
        break;
    case 1:
        SetScaleOfParticles(1.2f, colorChangeTime);
        break;
    case 2:
        SetScaleOfParticles(0f, colorChangeTime);
        break;
}
```

De functie SetScaleOfParticles() geeft weer hoe snel de switch van kleuren moet gebeuren en zet de groote van de particles.

De functies TryToAttack() checkt wanneer de speler dichtbij genoeg bij de pillars staat om te schieten. Als de speler te ver af zit is het voor de speler dus niet mogelijk om “raak” te schieten (ookal lijkt het wel zo). Als de speler wel dichtbij genoeg staat word de functie BeamAttack() uitgevoerd. Deze functie zorgt ervoor dat de speler word aangevallen door een beam die zn health omlaag gooit.

De functie “CheckIfPlayerInRange()” checkt of de speler dichtbij genoeg staat. Als hij dichtbij genoeg staat en een pillar nog niet dood is word de orb rood anders (dus als de pillar gestorven is) word de orb grijs.

De functie DrawLineToPlayerIfInRange() creeert een rode “enemy” line/light. Als je dichtbij genoeg een van de pillars staat komt er een soort laser straal aan je vast te zitten die je overal naar toe volgt. Deze laser straal schiet van tijd tot tijd een attack naar je (TryToAttack()) die je health omlaag gooit (BeamAttack()).

MovingBox.cs

Beschrijving

Dit script is onderdeel van de Ancient Slide Puzzle van level3 eiland 5. We handelen in dit script naar welke kant de box moet bewegen. Ook handelen we hier wat er gebeurd als de box botst tegen een paarse tegel of tegen een groene tegel.

Theorie & Ontwerpkeuzes

De box bestaat uit 6 kanten (net als een dobbelsteen). De functie OnHit() checkt door middel van de string “InComingFacade” welke kant van de box een collision heeft gehad. De functie OnHit() geeft dan de functie SlideBox() aan naar welke kant de box moet gaan bewegen.

De functie SlideBox() gebruikt dan de Switch-Case methode om de box een bepaalde richting op te duwen.

Wanneer de speler de doos verkeerd beweegt en de doos op een paarse/rode tegel land word de “OnDestroy()” functie afgespeeld. Deze functie zorgt ervoor dat de positie van de box word gereset en je het spel opnieuw moet gaan spelen.

De functie “FinishPuzzle()” word alleen uitgevoerd wanneer de speler het blok goed verplaatst heeft en het groene tegeltje bereikt heeft. Wanneer dit gebeurt wordt een deur naar volgende eiland geopend en kan de speler door lopen.

Activiteit - Debug Console

Welke scripts spelen een rol in deze activiteit?
CheatDebugController.cs, DebugCommand.cs

Wat voor gelinkte scripts zijn er?
N/A

referentie naar flowchart
Zie *Debug Console* gedeelte.

Korte beschrijving van de activiteit

Om ontwikkeling makkelijker maken moeten er manieren zijn om "cheats" aan te zetten zoals het verhogen van de rensnelheid, of het aanpassen van player health. Dit moet in elke level gedaan moeten worden.

Theorie

Onderzoeksvraag

Hoe kunnen we in elk level makkelijk bepaalde eigenschappen zoals player health aanpassen op een consequente manier.

Gevonden theorie & Resultaten & conclusie

Het beste is om een debug console toe te voegen. Deze kan je dan in elk level openen en zal het boven alle andere UI elementen verschijnen. Hierdoor kunnen we op een simpele manier in elk level commandos intypen die dan een bepaalde effect geven.

Door gebruik te maken van de Unity **OnGui** event functie kunnen we de UI in puur code schrijven en hoeven we geen extra objecten aan te passen.

Om het toevoegen van commands makkelijk te maken zijn er ook classes gemaakt die een debug command kunnen representeren.

CheatDebugController

Beschrijving

Dit is de hoofd script die de console functionaliteit heeft.

Theorie & Ontwerpkeuzes

Het hoofddoel van deze class is om de console UI te tekenen op het scherm wanneer het open staat. Het biedt functionaliteit voor het typen van een commando en het vervolgens uitvoering daarvan.

Wanneer een gebruiker op **Enter** drukt word **OnReturn()** uitgevoerd. Deze method kijkt of de gebruiker 'help' ingetypt heeft en zoja, zal het een lijst van alle beschikbare commandos. Als dit niet het geval is zal het de input doorgeven aan **HandleInput()**.

Deze method zal bij elke commando kijken of het de user input matched, en zoja zal het de commando uitvoeren.

Alle commandos worden gerepresenteerd door de **DebugCommand** of **DebugCommand<T>** class. De instanties zitten in de **commandList** list.

DebugCommand<T>

Beschrijving

Deze class representeert een Debug command die kan worden uitgevoerd.

Theorie & Ontwerpkeuzes

Bij een commando zijn er de volgende eigenschappen:

- Commando naam (bv set_health)
- Commando beschrijving
- Commando actie

Binnen deze class moeten deze informatie erin staan. De actie can de commando is wat uitgevoerd moet worden wanneer deze commando gebruikt wordt.

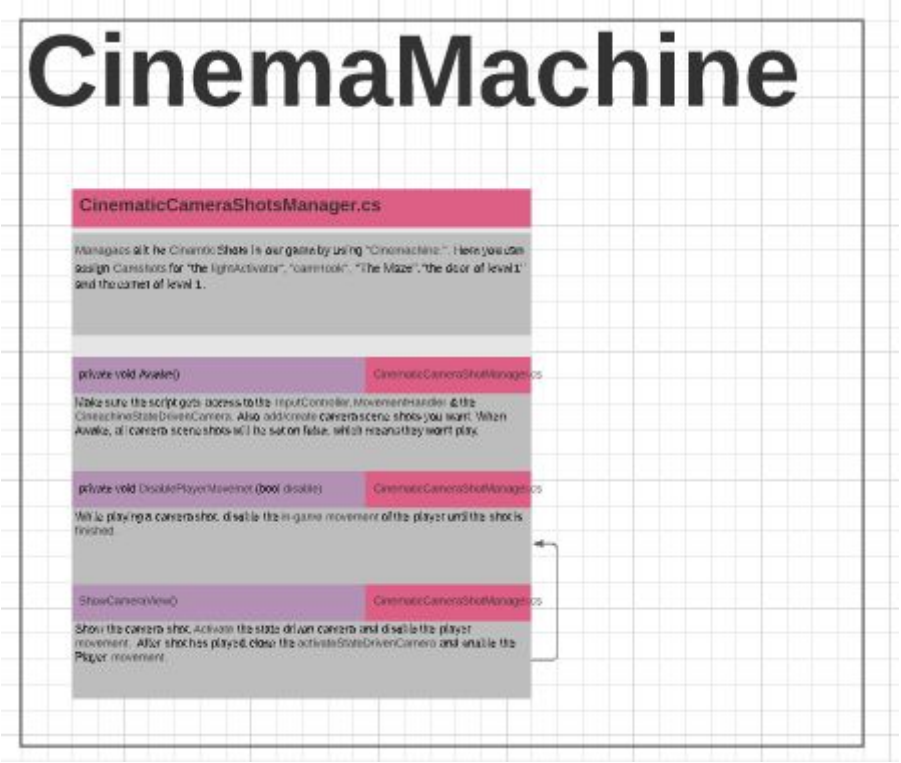
Bij het uitvoeren van een commando kan er ook een extra parameter meegegeven worden. De type van deze parameter is **T.** Als je commando geen extra parameter nodig heeft kan je de **DebugCommand** class gebruiken zonder de generieke type **T**.

Activiteit - CinematicCameraShotManager

Welke scripts spelen een rol in deze activiteit?
CinematicCameraShotManager.cs

Wat voor gelinkte scripts zijn er?
ShrineInteractable.cs - InputController.s - OnLightPuzzleAreaTrigger.cs - TutorialManager.cs - InputController.s - Movementhandeler.cs - CinemachineStateDrivenCamera.cs

Referentie naar flowchart



Korte beschrijving van de activiteit

Deze script manages alle Cinametic Shots die onze game heeft door het gebruik van “CinemaMachine”. We gebruiken dit bij alle korte “filmpjes/shots”, zoals de komeet die valt in Level1 of een deur die opengaat bij andere levels.

CinematicCameraShotManager.cs

Beschrijving

Creëert kleine shots / trailer oefen tekst in ons spel.

Theorie & Ontwerpkeuzes

CinemaMachine maakt gebruik van een Awake() functie. Deze functie zet de toegang tot de scripts van InputController, Movementhandeler & CinemachineStateDrivenCamera. Ook kan je vanaf hier scenes toevoegen of creeeren als je dat wilt. In de Awake() functie word bepaald welke van de al bestaande scenes als eerste word laten zien en welke nog in de wacht staan.

De functie “DisablePlayerMovement()” spreekt voor zich. Wanneer er een Shot word afgespeelt kan het zo zijn dat de gebruiker nog steeds op de pijltoetsen aan het drukken is om te blijven lopen. Met deze functie zetten we het lopen van de speler uit. Het lopen gaat weer aan wanneer de shot voorbij is.

De functie “ShowCameraView” laat de Camera Shot zien aan de speler. Het activeert de State Driven Camera en roept de DisablePlayerMovement() functie op zodat de speler gedurende de shot niet kan bewegen. Het maakt gebruik van de parameters “CameraParameterID”, wat inzich de ID van de scene die je wilt showen met zich mee draagt. Ook heeft het als parameter “timeBetweenSwitch” wat aangeeeft hoelang een switch duurt en een “hintMessage”, die een message weergeeft tijdens de shot. Dit kan bijvoorbeeld een stukje tekst zijn die zegt: “De deur is nu geopend!”

```
//Show camera scene
foreach (var cam in cameraParameterId)
{
    camTween = DOTween.Sequence();
    camTween.Append(DOTween.To(() => time, x => time = x, timeBetweenSwitch, timeBetweenSwitch));
    foreach (var allCams in allCamParameters)
    {
        if (allCams != cam)
        {
            camAnimator.SetBool(allCams, false);
        }
    }

    ActivateNeededVirtualCamAndDisableTheRestOfVirtualCams(cam);
    camAnimator.SetBool(cam, true);

    if (hintMessage != null)
    {
        SceneContext.Instance.hudContext.DisplayHint(hintMessage[counter]);
    }

    counter++;
    //yield return new WaitForSeconds(timeBetweenSwitch);
    yield return camTween.WaitForCompletion();
    camAnimator.SetBool(cam, false);

}
```

Verantwoording Keuzes technisch platform

Unity
We hebben als groep gekozen voor de Unity engine. De reden hiervoor is dat de minor is gericht op Unity en dat er veel workshops zijn gegeven in Unity2D/Unity3D, wat we goed konden gebruiken. Daarnaast hebben de developers ook ruime ervaring met Unity. In de loop van het project hebben we steeds meer kennis opgedaan op het gebied van Unity, hierdoor zijn we niet geswitcht van engine zoals bijvoorbeeld Unreal. Sommige developers willen ook graag doorgaan met Unity. Dit heeft ook een rol gespeeld in onze keuze.

Github
We hebben voor Github gekozen omdat we als team al ervaring mee hadden en al eerder gewerkt mee hebben.
Tijdens het schrijven van programmacode in 14 weken tijd is het belangrijk om versiebeheer goed bij te houden en te controleren. Om dit in goede banen te leiden hebben we een main branch aangemaakt voor de release van de game. Tijdens het ontwikkelprocess hebben we gebruik gemaakt van de “dev” branch. Dit is de centrale branch waar alles hoort te werken. Elke ontwikkelaar maakt een eigen branch aan vanuit de “dev” branch en merged dat weer naar de dev branch na het voltooien van de taak. Zo blijft de “dev” branch altijd up-to-date met de nieuwste functionaliteiten. Begin van de dag wordt er meestal gemerged naar je eigen branch zodat ieder de nieuwste versie heeft van de game. Dit zorgt voor minder conflicten en makkelijker werken. Dit geldt ook voor de artists en de designers.

Visual studio
De code is geschreven in C#. Aangezien Unity gebruik maakt van C# hebben wij ook hiervoor gekozen. Daarnaast hebben de developers ook eerdere ervaring met C# in Visual Studio.

DOTween
DOTween is een animatie engine in Unity, geoptimaliseerd in C#. Het is een handige tool om snel animaties te maken en heeft veel andere functies. We hebben voor deze tool gekozen omdat developers al eerder ervaring mee hadden en dit goed konden gebruiken.

Mirror
In de eerste weken van dit project hebben we een brainstormsessie gehouden met als eindresultaat dat een co-op game zouden ontwikkelen. Hiervoor hebben we de Mirror networking gebruikt. Dit is een open source networking api voor Unity. Na een gesprek met de docenten hebben ze ons geadviseerd om de co-op te laten vallen, doordat ze meer kennis hebben over de netwerk gedeelte en uit eerdere ervaring spreken. Uiteindelijk hebben we als team gekozen om een single player game van te maken.

Mapping TO met FO / GDD

Een helder overzicht waaruit duidelijk wordt waar welk onderdeel beschreven in het FO/ GDD in het TO wordt vermeld.

Story & Doelgroep

Het wereld staat op het punt om vernietigd te worden door een meteoor. De speler probeert zo snel mogelijk terug te komen bij zijn dorp. Terwijl hij naar zijn dorp gaat stort een deel van de meteoor. Hierdoor kan de speler de spirit wereld bekijken.

Bij dit gedeelte zal er meer over de doelgroep worden verteld. De game heeft een genre van een adventure puzzle game.

Meer informatie op pagina 3 van de GGD.

Gameflow

De game heeft in totaal 3 levels. De eerste level is een tutorial level waar wordt uitgelegd hoe je de game moet spelen. Daarnaast hadden we in het begin meer levels in gedachten maar, dit hebben we verminderd naar 3 levels, in verband met onze scope. Aangezien de game een puzzelgame is vergt het veel energie om de puzzels goed te programmeren wat ook veel tijd kostte.

Meer informatie op pagina 4 van de GGD

Character

Hieronder wordt beschreven wat voor karakter de game heeft, waar hij zich bevindt en wat zijn doel is. Met welke toetscombinaties de speler kan aanvallen, bewegen en springen. De speler heeft ook de mogelijkheid om te interacten met objecten en in de spiritwereld in te gaan. Door een interactable system te maken, kunnen de designers makkelijk de tekst wijzigen waarmee ze willen interacten.

In de spiritwereld zal de speler een shard-meter zien, die zal worden gebruikt in de spiritwereld. Ook kan de speler linksonderaan het scherm zien hoeveel levens hij nog heeft.

Meer informatie op pagina 5 van de GGD.

Gameplay

Om het beeld zo goed en mooi mogelijk te laten zien, is er er veel tijd besteed aan de gameplay. Het implementeren van het switchen van de werelds vergde veel tijd in, wat een belangrijke onderdeel is van onze game.

Meer informatie op pagina 6 van de GGD.

Core mechanics

De pijl en boog en de spirit state zijn een van belangrijkste mechanics in de game. Om dit vloeiend te laten verlopen en in de game te implementeren was er goede communicatie tussen de artists en de developers.

Meer informatie op pagina 8 van de GGD.