Pixel Potato

Technische onderbouwing

Get Off Me

Niek Oosterbaan, 0884141 & Swen Meeuwes, 0887127
Laatst geüpdatet op: 8-10-2017

Inhoudsopgave

Moeilijkheidsgraad - Game aspecten

Medium - Online output (Google play service)

Voor de leaderboards (en misschien wel toekomstige achievements) is er voor gekozen om gebruik te maken van 'Google Play Services'. Dit platform biedt ons de mogelijkheid om leaderboard, achievements en events te beheren. Verder is de beheerder van het platform (Google) betrouwbaar.

De speler kan hier gebruik van maken zodra deze een account heeft bij Google Play Services. Als de speler deze al niet heeft natuurlijk! Dit is al gauw het geval omdat Google Play Services door meerdere games op de Play Store gebruikt wordt.

Medium - Physics

In het begin was het physics system nog custom gemaakt, het was toen nog erg simplistisch. De movement van de slimes naar de player ging volgens de volgende formule; Pf = Pi + vt +½at2. Waarbij Pf = positie, Pi = initiële positie, v = velocity, t = time en a = acceleration. Echter toen er elementen zoals wrijving toegevoegd moesten worden is de keus gemaakt om over te stappen naar het physics systeem van Unity. Dit systeem verving toen het oude systeem en had als extra al een wrijving functionaliteit ingebouwd.

Medium - Collectables

Om de moeilijkheid van het spel op te krikken is er een systeem gekomen waarbij er collectables geactiveerd kunnen worden als risk/reward; Vials. Deze zijn geïmplementeerd als volgt: Er is een interface “IVial” die elke concrete class van een vial implementeerd. In deze interface staan meerdere “Apply()” functies die elk een parameter heeft van de verschillende soorten type objecten die een vial kan modificeren. Zo kan het object dat gemodificeerd moet worden aan de vial gegeven worden, en kan hij aangepast worden binnen de concrete class van een vial. Om dit te laten werken is er gebruik gemaakt van het “Visitor” design pattern. Elke concrete class die aangepast kan worden door een vial moest een functie “Accept” implementeren die de Apply functie aanroept. Hier geldt de ‘Apply’ methode als de ‘Visit’ call en de concrete vials als de visitors.

Dit ziet er als volgt uit in code:

IVial.cs

```
public interface IVial {
        void Apply(HelmetSlimeEnemy entity);
        void Apply(NormalSlimeEnemy entity);
        void Apply(RogueSlimeEnemy entity);
        void Apply(WizardSlimeEnemy entity);
        void Apply(MedicSlimeAlly entity);
        void Apply(BombSlimeEnemy entity);
        void Apply(Player player);
        void Apply(ComboSystem comboSystem);
        void Apply(OffScreenSpawner spawner);
        void Apply(List<GamePhase> phases);
}
```

SpeedVial.cs

```
public class SpeedVial : IVial {
        private const float speedModifier = 0.7f;
        private const float doubleComboChanceModifier = 30;

        public void Apply(HelmetSlimeEnemy entity){
                entity.Model.speed += speedModifier;
        }
        public void Apply(NormalSlimeEnemy entity){
                entity.Model.speed += speedModifier;
        }
        public void Apply(RogueSlimeEnemy entity){
                entity.Model.speed += speedModifier;
        }
        public void Apply(WizardSlimeEnemy entity){
                entity.Model.speed += speedModifier;
        }
        public void Apply(MedicSlimeAlly entity){
                entity.Model.speed += speedModifier;
        }
        public void Apply(BombSlimeEnemy entity) { }

        public void Apply(Player player){}
        public void Apply(ComboSystem comboSystem){
                comboSystem.chanceAtDoubleCombo += doubleComboChanceModifier;
        }
        public void Apply(OffScreenSpawner spawner) { }
        public void Apply(List<GamePhase> phases) { }
}
```

Accept methode in concrete class

```
public override void Accept (IVial vial) {
        vial.Apply(this);
}
```

Hard - Infiniteness

De insteek van het design was bepalend dat het een infinite spel zou worden. Het stress-effect wat men ervaart wanneer de speler groeit heeft vooral geholpen in dit besluit. Met een level systeem zou dit effect veel minder voorkomen.

De implementatie van een infinite level was goed te doen. Er is gekozen voor een “phase” systeem. Dit houdt in dat er per “phase” ingesteld kan worden wat voor soort vijanden/allies spawnen en hoe vaak. Wanneer de laatste “phase” bereikt is, blijft hij deze gebruiken om vijanden/ allies te spawnen. De gamedesigner heeft dan vanaf dit punt de keus om de spawn snelheid lineair of stijgend te maken. De grootste uitdaging hierin was voor de designer om een goede balans te vinden tussen moeilijkheidsgraad en plezier. Om deze game balans makkelijk te veranderen zijn ook hiervoor editors gekomen, zie “Hardcore - Level editor”.

Hard - Combo system

Het combo systeem is uiteindelijk zo gemaakt dat spelers meer rewards krijgen voor het dichterbij laten komen van vijanden. Een risk/ reward factor dus. Dit is gedaan om de veteraan spelers toch een uitdaging te geven en het voor de casual spelers nog wel haalbaar te houden.

Het werkt als volgt: Er is een cirkel aangegeven voor de speler. Wanneer een vijand binnen de cirkel weg geswipet wordt, krijgt men +1 combo. Per extra combo krijgt men een variabele multiplier op je score die je krijgt per kill. Na X aantal combo’s gaat men de volgende “combo tier” in. Dit zorgt ervoor dat de cirkel waarin je de combo ontvangt, verkleind wordt.

Hardcore - Level Editor

Bij het ontwikkelen van dit project hebben we verschillende tools aangeboden om het spel makkelijk te kunnen tweaken. Zo kon de game designer efficiënt te werken gaan!

Verder was dit niet alleen fijn voor de game designer maar ook voor de programmeurs, omdat dit de standaard was werd alles gelijk dynamisch opgezet.

Voorbeelden van de editors zijn:

- Tutorial sequence editor
    - Hierin kan de tutorial sequence aangepast worden door verschillende stappen te definiëren. De stappen kunnen later makkelijk toegevoegd, verplaatst, bewerkt en verwijderd worden.
- Spawner editor
    - Geeft toegang tot de spawn intervallen en snelheid van de spawns. Deze zijn afgezet tegenover de speeltijd en kunnen aangepast worden in een curve.
- Player editor
    - In deze editor kan de hoeveelheid levens en de shockwave ability aangepast worden.
- Enemy editor
    - Hierin kunnen vijand gerelateerde kenmerken aangepast worden. Voorbeelden hiervan zijn:
        - Gewicht
        - Levens
        - Snelheid
        - Random snelheid offset
        - Het aantal punten dat ze waar zijn
- Vial Context
    - Hierin staat wat iedere vial (difficulty modifier) doet en wat de unlock conditions zijn.
- Phase editor
    - Hierin kunnen de soort vijanden die spawnen bepaald worden. Er moet een tijd ingesteld worden vanaf hoeveel seconde de phase ingaat, en de “weights” van alle enemies. Dit werkt als volgt: een entity met weight 6 spawnt 1.5x zo vaak als een entity met weight 4. Dit “weight” systeem zorgt ervoor dat het makkelijker is om nieuwe entities toe te voegen en om de spawn rates in runtime aan te kunnen passen. Dit gebeurd bijvoorbeeld bij één van de vials, die ervoor zorgt dat er meer medics spawnen (+1 weight voor medics in elke phase).
    - Wanneer men een nieuwe entity toevoegt in deze interface, een prefab selecteert en weights invult, zal deze meteen meegenomen worden in de game.

Hardcore - Gesture-based controls

Om de speler te belonen voor het halen van hoge combo's is de 'shockwave' ability geïntroduceerd. Deze ability kan de speler activeren door middel van een 'zoom out' gesture.

Deze gestures (en mogelijk toekomstige gestures) worden afgehandeld in de 'Input Manager'. Hierin wordt gekeken of de speler het scherm aanraakt met 2 vingers (of meer). Als dit het geval is dispatched de Input Manager een event die de het verschil tussen de lengte van de 1ste vector en 2de vector bevat.

```
public static string GESTURE_DETECTED = "Gesture detected";

private void HandleZoomGesture()
{
        if (Input.touchCount < 2)
            return;

        var touchOne = Input.GetTouch(0);
        var touchTwo = Input.GetTouch(1);

        var previousTouchOnePosition = touchOne.position - touchOne.deltaPosition;
        var previousTouchTwoPosition = touchTwo.position - touchTwo.deltaPosition;

        var previousTouchDelta = (previousTouchOnePosition - previousTouchTwoPosition).magnitude;
        var touchDelta = (touchOne.position - touchTwo.position).magnitude;

        var touchDeltaDifference = previousTouchDelta - touchDelta;

        var eventObject = new PinchGesture() {
            DeltaDifference = touchDeltaDifference,
            TouchOne = touchOne,
            TouchTwo = touchTwo
        };

        Dispatch(GESTURE_DETECTED, eventObject);
}
```

Gameplay entity abstraction