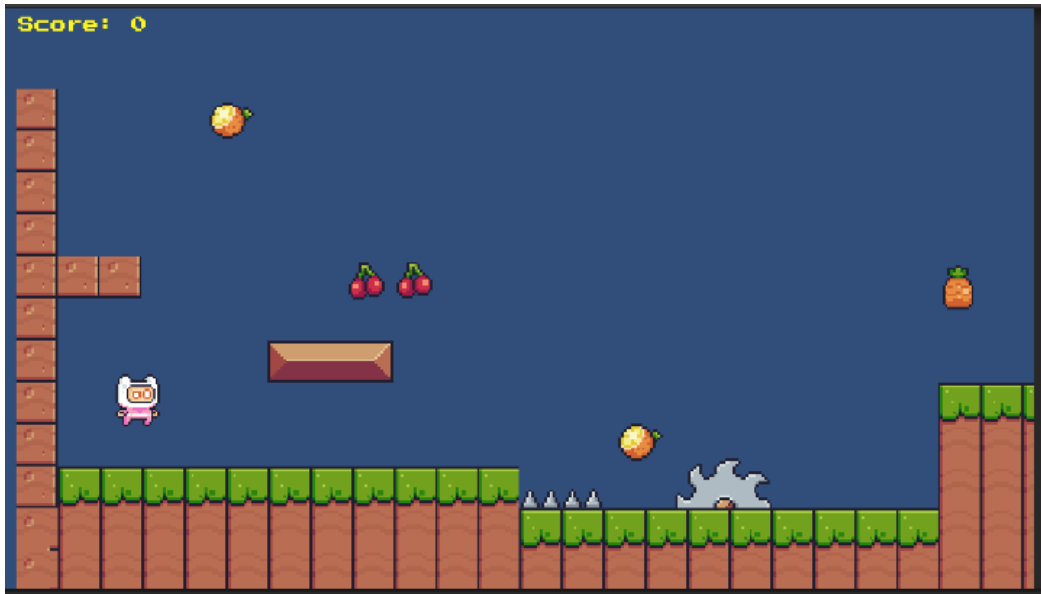


Unity Intro

Af: Michael Hansen, Coding Pirates Furesø, 2022, version 0.50

Dokument og kode ligger her: <https://github.com/mhfalken/unity/>



Dette er en tutorial i hvordan man laver et lille 2D platform spil i Unity, som vist på billedet. Det inkluderer alt fra hvordan man installerer Unity, til en start pakke og til en fuld gennemgang i hvordan man gør. Der er ingen specifik C# undervisning, men det meste af den kode som man skal bruge er vist med eksempler. På min github ligger den fulde "løsning" som en pakket fil (TBD). Dokumenter indeholder også i bunden nogle links til gode videoer.

1	Installation af Unity på PC/MAC.....	4
1.1	Installation af C# editor	4
2	C# editor mini guide (Visual Studio)	5
3	Tutorial	7
3.1	Tegn banen	7
3.2	Tilføj player	7
3.3	Flyt Player	8
3.4	Hop Player	9
3.5	Animationer	10
3.6	Frugter	12
3.7	Lyd	13
3.8	Game tekst og point	14
3.9	Simple forhindringer	14
3.10	Bevægelige forhindringer	15
3.11	Udvid banen til flere 'skærme'	15
3.12	Avanceret animationer	15
3.13	Tænd/sluk forhindringer	15
3.14	Flere baner.....	15
4	Unity Reference	16
4.1	Assets foldere	16
4.2	Eksterne felter	16
4.3	Komponenter.....	17
4.4	Taster	17
4.5	Bevægelse.....	17
4.6	Kollision	18
4.7	Animation	19
4.8	Lyd	20
4.9	Game tekst.....	20
4.10	PreFabs	21
4.11	Partikler	21
4.12	Dynamisk opret og slet et objekt.....	21
4.13	Script kommunikation	22
5	Utility	22
5.1	Tidshåndtering.....	22
5.2	Fejlsøgning.....	22
5.3	Interval check	23
5.4	Forsinkelse.....	23
5.5	Importer Unity pakke	23
6	Specielle scripts	23

6.1	Bevægelige objekter	23
6.2	Stå fast på platforme som bevæger sig	25
7	Special setup.....	25
7.1	Flyt kamera efter spiller.....	25
7.2	Glat overflade	25
7.3	Lyd support (avanceret)	26
8	Links.....	26

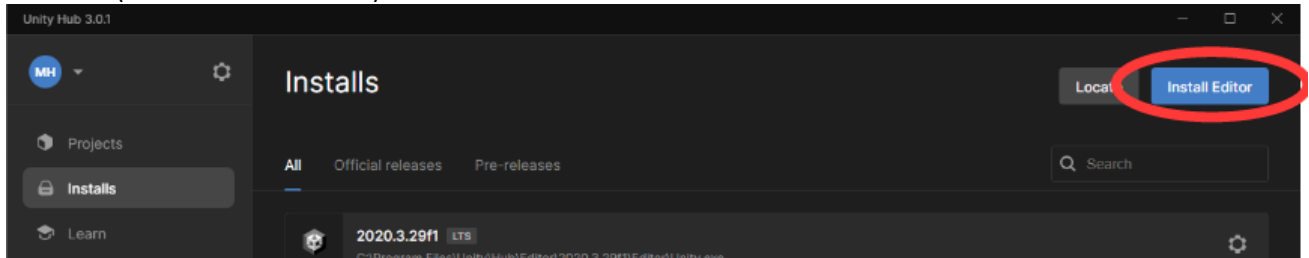
1 Installation af Unity på PC/MAC

Man skal bruge en nyere PC/MAC med 10GB ledig disk plads.

Man installerer Unity ved at downloade og installerer Unity Hub først:

<https://unity.com/download>

Efter at man har startet Unity Hub'en skal man oprette en bruger og det kræver en email adresse som man har adgang til. Når man har oprettet sin bruger skal man logge ind på sin Unity Hub og installere selve Unity Editoren (Installs: Install Editor).



Her skal man vælge en *officiel release* fx 2020.3.xxxx (**LTS**). Den fylder ca. 6GB og det tager derfor noget tid at hente afhængig af internetforbindelse og computeren. Når den er downloadet, så installeres den automatisk. Det tager normalt ca. 10 minutter.

1.1 Installation af C# editor

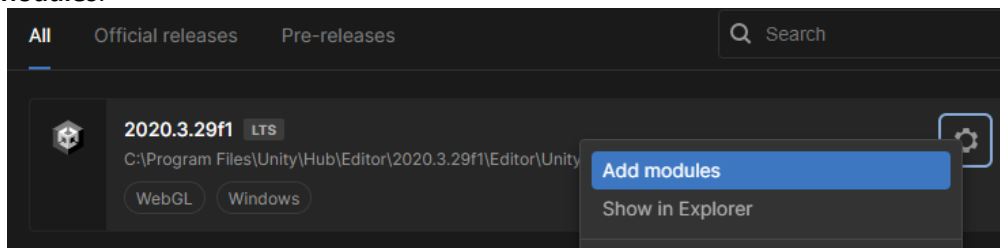
Nu skal man **vælge** hvilken C# editor man ønsker at bruge. Man kan reelt vælge helt frit, men det vil være en stor fordel at vælge en af de to følgende:

- Microsoft Visual Studio (kan installeres indefra Unity HUB)
- Visual Studio Code + C# plug-in

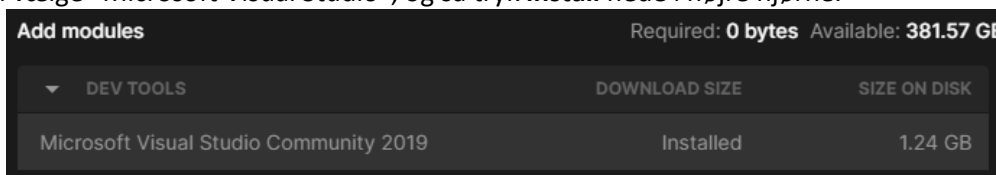
MVS kan give lidt problemer at installere og kræver efterfølgende at man opretter en Microsoft konto, men til gengæld er det den officielt supportet version og har lidt flere features. Begge vil dog virke fint og ser ens ud, så her er en vejledning til begge to.

1.1.1 Installer Microsoft Visual Studio

Man går ind i Hub'en og finder den release som man har installeret og trykke på 'tandhjulet' ude til højre og vælge **Add Modules**.



Her skal man vælge "Microsoft Visual Studio", og så tryk **install** nede i højre hjørne.




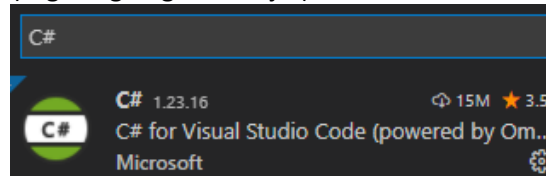
Indenfor 30 dage skal man logge ind på sin Microsoft Visual Studio editor med en Microsoft konto ellers løber demolicensen ud og MVS holder op med at virke. Dette gøres indefra MVS editoren under:

Help->Register Visual Studio.

1.1.2 Installer Visual Studio Code

Man skal installere følgende fil: <https://code.visualstudio.com/>

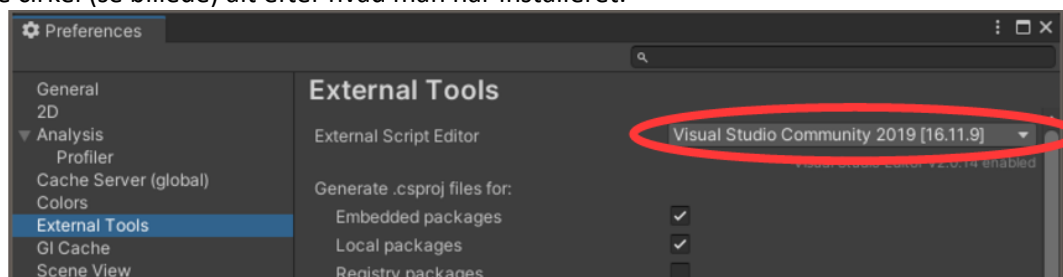
Derefter skal man installere en **C#** plugin. Åben VSC og tryk ude til venstre på  for at åbne **extensions**. I søgefeltet skriv: C# (med stort C) og vælg følgende linje (er normalt den øverste):



og installer den.

1.1.3 Vælg C# editor i Unity (gælder begge to)

Første gang man starter et Unity projekt, skal man gå ind i **Edit->Preferences->External tools** (MAC ???->**Preferences->External tools**) og vælge C# editor "Visual Studio Community 2019" eller "Visual Studio Code" i den røde cirkel (se billede) alt efter hvad man har installeret.



2 C# editor mini guide (Visual Studio)

Den C# editor (det program som man skriver sin C# kode i) hedder i det følgende MVS. Det gælder også selvom man bruge Visual Studio Code, da de ser ens ud.

Den kan en masse forskellige ting, hvor nogle få af dem er beskrevet her.

MVS "kommunikere" med Unity og kender dens syntaks. Den kan derfor hjælpe en når man skriver sin kode.

- Hvis man starter med at skrive et navn på en funktion som den kender, så viser den løbende nogle forslag. Her kan man vælge fra en liste eller bare trykke på **Enter** hvis den 'rigtige' allerede er valgt.
- Hvis man skal lave fx en **for** loop, så man kan skrive **for** og trykke på **TAB** tasten, så laver MVS automatisk en **for** loop skabelon. Dette virker for: **for/while/if/foreach/switch**. Det gør det lettere, hvis man ikke er helt sikker på syntaksen.
- Man kan få hjælp til en Unity funktion ved at holde musepilen over navnet.
- MVS farver alt afhængig af hvad det er, så kik på farverne for at se om det ser rigtigt ud.
- Hvis noget er understreget med rødt, så er det fordi MVS mener der er noget galt.
- Unity GUI 'ser' ikke rettelsen før filen er gemt.

Gode genvejstaster: (CTRL = CMD på MAC)

Tast	Funktion
CTRL-S	Gem filen
CTRL-Z	Undo
CTRL-C	Kopier
CTRL-V	Indsæt
CTRL-H	Søg/erstat

Specielle MAC taster:

Tast	Funktion
ALT-8	[
ALT-9]
SHIFT-ALT-8	{
SHIFT-ALT-9	}

3 Tutorial

Denne tutorial vil gennemgå hvordan man laver et 2D Platform spil – se forsiden af dette dokument.

Åbent Unity HUB'en og opret et nyt projekt ved at trykke på **New Project** i øverste højre hjørne. Vælg **2D** og giv det et navn, fx **cpf_spil** og tryk **Create project**. Det tager lidt tid ... Det program som kommer frem vil fremover blive kaldt **GUI'en**.

Det først man skal gøre er at importere den start pakke vi skal bruge. Den ligger her:

https://github.com/mhfalken/unity/blob/main/cpf_start_pakke.unitypackage

Vælg download og gem filen et sted du kan finde igen.

I GUI'en: **Assets->Import Package->Custom package** og vælg ovenstående fil. I det vindue som kommer frem trykkes på **Import** i nederst højre hjørne. Pakken er nu importeret og ligger under **Assets**.

Vælg **Assets/Levels** og dobbelt tryk på **Level1**. Slet **Assets/Scenes**.

Lige nu skal der i **Scene** vinduet gerne være 6 grønne felter.

3.1 Tegn banen

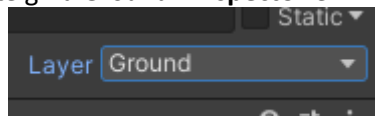
Tegn en lille simpel bane med mindst en platform som man kan hoppe op på (det skal ikke ligne min!). Man kan altid tegne videre senere...

Tryk på **Terrain** i **Hierarchy**'et (under **Grid**) og derefter på **Open Tile Palette** i **Scene** vinduet. Ved at vælge billederne i **Tile Palette** vinduet kan man tegne sin bane i **Scene** vinduet. Man kan også slette, kopier mv. ved at vælge i denne menu (øverst i **Tile Palette** vinduet).



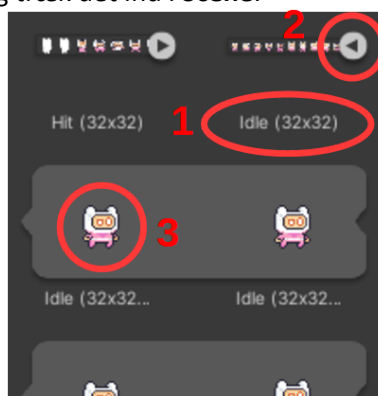
Når man er færdig med at tegne, skal man i **Terrain Inspector**'en sætte **Layer** til **Ground**. Først skal **Ground** oprettes, det gøres ved at trykke på den lille pil ned og vælge **Add Layer**. Skriv **Ground** ud for **User Layer 3**.

Tryk på **Player** i **Hierarchy**'et igen og vælg nu **Ground** i **Inspector**'en.

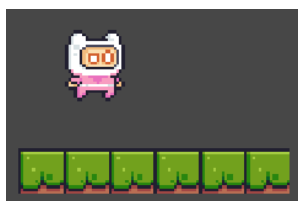


3.2 Tilføj player

Vælg en figur type under **Main Characters**. Tryk på billedet af **idle** (1) versions pil til højre (2). Tryk på det første billede (3) som kommer frem og træk det ind i **Scene**.

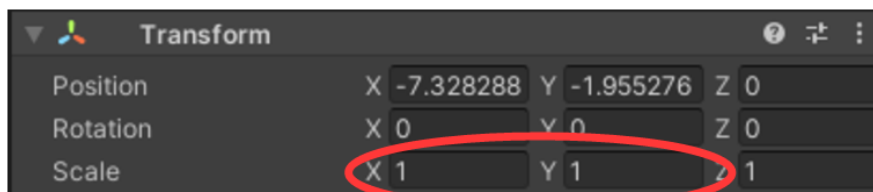


Det skal nu gerne se sådan ud.



Der kommer også et nyt objekt ind i **Hierarchy**'et som hedder noget med "Idle ...". Rename nu 'Idle ...' til **Player**. Husk stort P!


Man kan ændre størrelsen af figuren i **Inspector**'en ved at ændre **Scale** værdierne.



Tryk på **Play** i toppen af skærmen og se hvordan det ser ud. Der sker ingen ting, Player bliver 'flyvende' i luften.

Vælg *Player* i **Hierarchy**'et og i **Inspector**'en (ude til højre) tryk **Add Component** og vælg **Rigidbody 2D**.

Tryk **Play** og se hvad der sker! Nu falder Player gennem platformen. Det skyldes at den ikke har nogen Collider. I **Inspector**'en tilføj nu en **Box Collider 2D** og tryk Play igen. Nu skal figuren gerne falde ned og stå på platformen.

Tilpas **Collider**'en så den bedre passer til størrelsen af figuren ved at trykke på **Edit Collider** . Man skal under Rigidbody 2D sætte følgende (ellers kan figuren vælte):



3.3 Flyt Player

For at få Player til at flytte sig skal vi oprette vores første script. I folderen **Assets/Scripts/** højreklik og opret et **Create->C# Script** og kald det *PlayerController*. (Det er vigtigt at gøre det i et flow – man må ikke rename filen ...) Træk nu dette script op over *Player* i **Hierarchy**'et.

Dobbelt klik på scriptet for at åbne det i MVS editoren. Koden her er skrevet i C# og det er meningen at man selv skal tilføje den kode som mangler alt efter hvad man vil opnå.

Hvis man ønsker en gennemgang af C# er her en link til en god guide på dansk, lavet specielt til Unity:

<https://github.com/Grailas/CodingPiratesAalborg/blob/master/Guides/Hj%C3%A6lpeguide.pdf>

Skriv følgende linjer i `Update()` funktionen:

```
void Update()
{
    float dir = Input.GetAxisRaw("Horizontal");
    Debug.Log(dir);
}
```

Kik på **Console** vinduet og tryk på **Play**. Tryk nu på *piletasterne* og se hvad der sker. Debug linjen skriver ud hver gang man trykker på en piletast. (Virker også med A og D). Det er linjen `Debug.Log()` som printer ud på **Consolen** og er et vigtigt redskab når man skal debugge sin kode.

Ændre nu filen til følgende:


```

public class PlayerController : MonoBehaviour
{
    [SerializeField] float speed = 10;

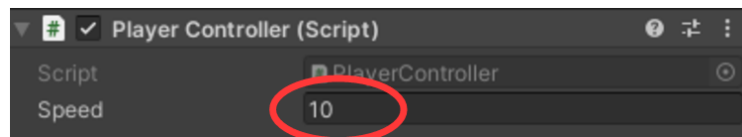
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float dir = Input.GetAxisRaw("Horizontal");
        transform.Translate(dir * speed * Time.deltaTime, 0, 0);
    }
}

```

Tryk på **Play** og se om det virker. Man kan ændre hastigheden i **Inspector**'en ved at ændre på **Speed** værdien.



3.3.1 Flip Player

Figuren skal nu kikke i den retning som den bevæger sig. Det gøres ved at bruge **Flip** feltet i **Sprite Renderer**. Dette skal selvfølgelig gøres automatisk fra *PlayerController* scriptet. Tilføj den manglende kode:

```

public class PlayerController : MonoBehaviour
{
    [SerializeField] float speed = 10;
    SpriteRenderer sr;

    // Start is called before the first frame update
    void Start()
    {
        sr = GetComponent<SpriteRenderer>();
    }

    // Update is called once per frame
    void Update()
    {
        float dir = Input.GetAxisRaw("Horizontal");
        transform.Translate(dir * speed * Time.deltaTime, 0, 0);
        if (dir < 0)
            sr.flipX = true;
    }
}

```

Nu kan figuren vende til venstre men ikke tilbage igen til højre. Tilføj selv de to linjer kode som mangler.

3.4 Hop Player

Vi skal nu have *Player* til at kunne hoppe og det gøres ved at tilføje noget mere kode til *PlayerController* scriptet. Selve hoppet virker ved at tilføje en kræft til **Rigidbody** komponenten:

```
rb.AddForce(Vector3.up * jumpPower);
```

Rigidbody komponenten skal først 'hentes' (ligesom SpriteRenderer blev det) og dette gøres på følgende måde:

```
Rigidbody2D rb;  
...  
rb = GetComponent<Rigidbody2D>();
```

jumpPower skal være en synlig variabel i **Inspector**'en, så man senere kan rette i den:

```
[SerializeField] float jumpPower = 200;
```

Jump tasten aflæses på følgende måde:

```
bool jump = Input.GetButtonDown("Jump");
```

Sæt nu linjerne ind de "rigtige" steder i scriptet og tilføj en **if** sætning for at udføre hoppet når man trykker på *Space* tasken. Juster hoppe højden i **Inspector**'en med værdien **Jump Power**.

3.4.1 Ground detekt (avanceret)

Man kan hoppen mens man er i luften, hvilket jo ikke er meningen. Det kan man undgå ved at checke om man står på jorden inden man hopper.

Følgende funktion undersøger om man står på jorden inden man hopper.

```
[SerializeField] LayerMask groundLayer;  
...  
bool GroundCheck()  
{  
    return Physics2D.CapsuleCast(bc.bounds.center, bc.bounds.size,  
                                0f, 0f, Vector2.down, 0.5f, groundLayer);  
}
```

Ground Layer feltet i **Inspector**'en skal sættes til *Ground*.

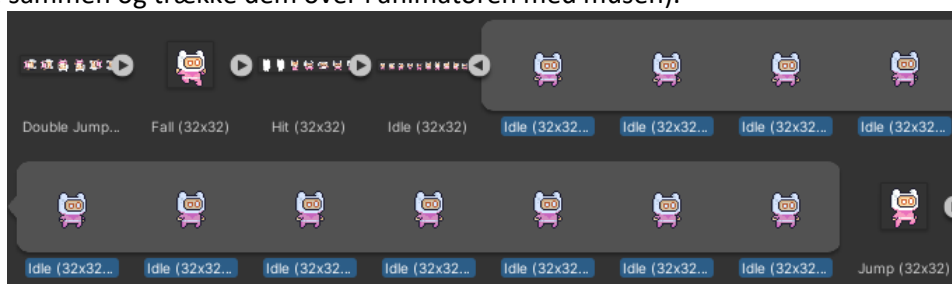
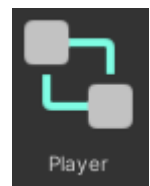
Brug funktion til at lave et ekstra check inden man hopper.

3.5 Animationer

Nu skal vi have animeret vores *Player*. Under den grafik folder hvor I har fundet jeres *Player (Main Characters/)*, findes nogle forskellige "animationer", dvs. serie af billeder som vist i den rigtige hastighed laver en animation. Vi skal starte med at lave en animation når *Player* står stille (idle).

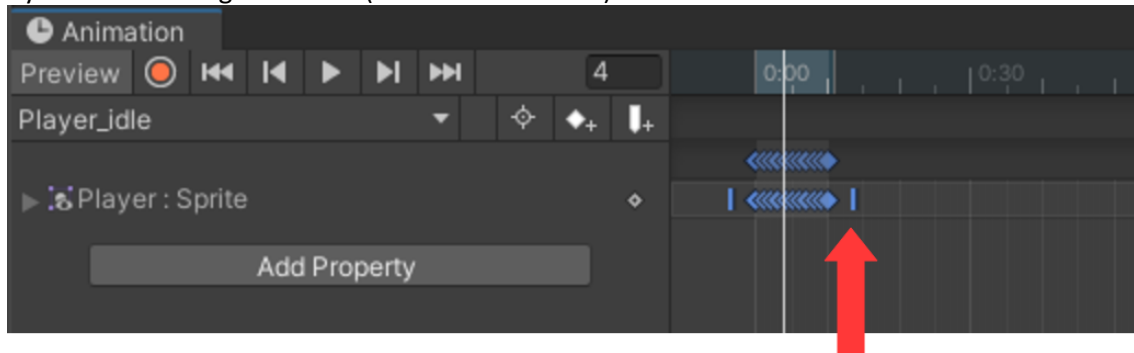
Start med at lave en folder under **Assets** som hedder **Animator**.

1. I folderen *Animator*, **Create->Animation** og kald filen noget 'fornuftigt', fx *Player_idle*.
2. Træk derefter *Player_idle* filen til objekt *Player* i **Hierarchy**'et. (Dette laver også en *Player controller* første gang man gør det – se billede.)
3. Så skal man åbne animationsvinduet: **Window->Animation->Animation**.
4. Vælg derefter *Player* objektet i **Hierarchy**'et.
5. Vælg nu alle de billeder som udgør animation og træk dem over i **Animatoren**. (Det gøres ved at finde den *Idle* animationsfil man vil bruge og trykke på den lille pil så man kan se alle billederne, vælge alle sammen og trække dem over i animatoren med musen).



Det skal se ud som på billedet neden for.

- Man kan nu se hvordan det ser ud ved at trykke på play knappen, og justere hastigheden ved at flytte den blå streg til siderne (se billede nedenfor).



- Hvis animation skal forsætte, skal man huske at sætte *Loop Time* i **Inspector**'en for *Player_idle*.

Når man er færdig med animationsvinduet, kan man med fordel dock'et det til et af de andre vinduer man har fx **Console** vinduet, så det er lettere at finde næste gang.

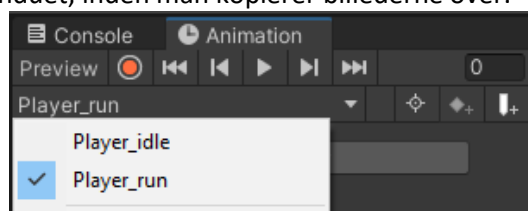
Prøv spillet og se om det virker.

3.5.1 Flere animationer (avanceret)

Dette kan laves senere, hvis man hellere vil videre.

Vores *Player* har brug for mere end en animation, da han også skal have en løbe animation (run).

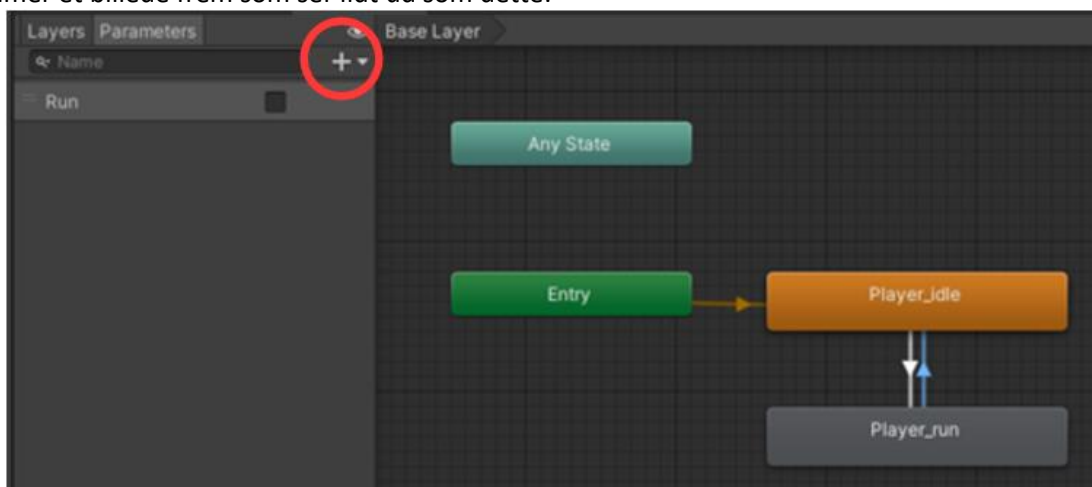
For hver ekstra animation man har brug for, gentager man skridtene oven for. Husk at man skal vælge den rigtige animation i **Animation** vinduet, inden man kopierer billederne over.



Da man nu har flere animationer, er man nødt til at fortælle Unity hvilken animation man skal bruge hvornår. Åben *Player Animator*'en – dobbelt klik på:



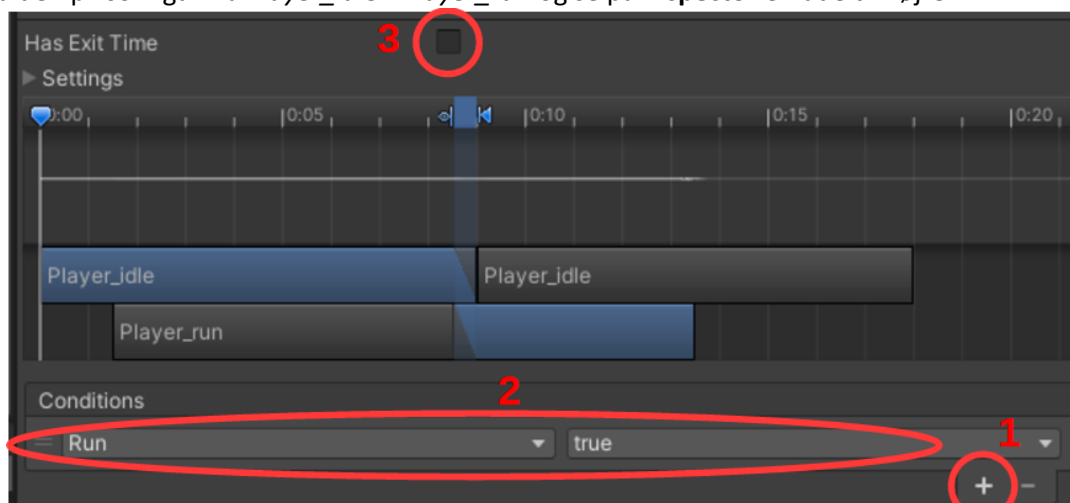
Der kommer et billede frem som ser lidt ud som dette:



Det er et diagram, som viser de forskellige animationstilstande og hvordan man kan komme fra den ene til den anden.

Lav de to lodrette pile ved at højre-klikke på *Player_idle*, vælg **Make Transistion** og træk den til *Player_run*. Gør det samme modsat også, så der er to pile – en i hver retning – se billede. Tilføj en parameter ved at klikke på '+' i toppen ud til venstre (rød cirkel), vælg **Bool** og kald den *Run*.

Klik nu på den pil som går fra *Player_idle*->*Player_run* og se på **Inspector**'en ude til højre.



Tryk på '+' [1], hvilket automatisk laver [2]. Fjern markeringen i **Has Exit Time** [3]. Gør det samme for den anden pil, men vælg denne gang **false** i [2].

Det er nu sat sådan op, at når *Run* er **true**, så skiftes til *Player_run* state og når *Run* er **false**, så skiftes til *Player_idle*.

Nu skal vi have vores *Player* scrip til at styre denne *Run* parameter. Tilføj følgende linjer til *PlayerController* scriptet (de rigtige steder).

```
Animator animator;
...

animator = GetComponent<Animator>();
...

if (dir == 0)
    animator.SetBool("Run", false);
else
    animator.SetBool("Run", true);
```

Hvis *dir* (vores bevægelse) er 0 (vi står stille), så sætter vi *Run* til **false** (dvs. løb ikke), ellers sæt *Run* til **true** (løb).

Prøv at se hvordan det virker.

3.6 Frugter

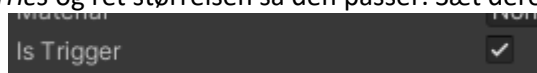
Nu skal vi have nogle ting vi kan 'tage'. Der er en række frugter, som kan bruges til dette. Frugterne skal også animeres, men her gør vi det på en lidt anden måde (lettere), da de kun skal have én animation pr. frugt. Under *Fruits* vælg *Cherries* og sæt **Pixels Per Unit** til 16 i **Inspector**'en. Tryk derefter **Apply** nede i højre hjørne. Dette skal generelt gøres for alle de grafikbilleder som man skal bruge. Jeg har gjort det for nogle få af dem, resten må I selv klare. Hvis man 'glemmer' det så bliver de bare meget små.



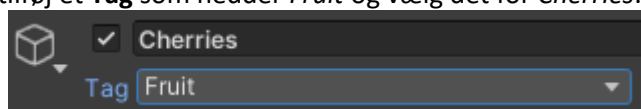
Tryk på *Cherries* og træk den ind i **Scenen**. I dialog boksen som kommer frem, gem filen i *Animator* folderen og kald den *Cherries*. Ret navnet i **Hierarchy**'et til *Cherries* (fjern _0).

Prøv at køre det og se hvordan bærret bevæger sig.

Tilføj en **Box Collider 2D** til *Cherries* og ret størrelsen så den passer. Sæt derefter **Is Trigger**.



En *trigger* kollision betyder at man ikke får en reel kollision, men kun en trigger på at de to figurer overlapper. I **Inspector**'en tilføj et **Tag** som hedder *Fruit* og vælg det for *Cherries*.



Tilføj følgende linjer i bunden af *PlayerController* scriptet (før den sidste '}').

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Fruit"))
    {
        Destroy(collision.gameObject);
    }
}
```

Test og se hvad der sker når *Player* rammer frugten.

3.6.1 Prefabs

Når man har lavet sin frugt færdig, skal man lave en Prefab. En Prefab er en skabelon som man kan bruge igen og igen.

Først laver man en folder under **Assets** som hedder **Prefabs**. Træk nu *Cherries* fra **Hierarchy**'et ned i **Prefabs** folderen.

Hver gang man vil tilføje endnu en frugt, så trækker man bare en fra **Prefabs** folderen direkte ind i **Scenen** hvor man vil have den. Prøv og tilføj nogle flere og se at det virker.

Tilføj nu nogle flere forskellige frugter og lav **Prefabs** for hver af dem.

Lav et **Create Empty** objekt i **Hierarchy**'et og kald det *Frugter*. Her gør Unity noget lidt irriterende – den sætter Z positionen til noget andet end 0! Ret det i **Inspector**'en.

Træk alle frugterne ind under dette new objekt for at få lidt mere orden.



3.7 Lyd

Spillet er lidt stille, så det er på tide at sætte lidt lyd på. Der ligger allerede nogle lydfiler i folderen **Assets/Sounds/**. Ved at dobbelt klikke på dem kan man høre hvordan de lyder.

For det *objekt* som man ønsker lydsupport for, skal man først tilføje en **AudioSource** komponent. Tilføj **AudioSource** til *Player* objektet.

I *PlayerController* scriptet, indsæt disse linjer de 'rigtige' steder i koden.

```
[SerializeField] AudioClip jumpSound;
...
AudioSource audioPlayer;
...
audioPlayer = GetComponent();
...
audioPlayer.PlayOneShot(jumpSound, 1);
```

Den første linje laver et felt i **Inspector**'en som man skal trække den lyd over i man ønsker at bruge til hop. Den sidste linje afspiller selve lydfilen, og skal stå der hvor man hopper.

Prøv om lyden virker når man hopper.

Man kan tilføje flere lyde ved for hver lyd at lave disse to linjer.

```
[SerializeField] AudioClip itemSound;
...
audioPlayer.PlayOneShot(itemSound, 1);
```

Den første linje bruges til at vælge hvilken lydfil man ønsker og den anden linje afspiller lydfilen.

3.8 Game tekst og point


Når man samler frugter og andre ting i spiller, skal man have point for det. Disse point skal selvfølgelig vises på skærmen.

Man starter med at indsætte en tekst i **Scenen** på følgende måde:

Lav et objekt i **Hierarchy**'et og vælg **UI->Text** og kald det **Score**. Det vil automatisk lave to objekter: **Canvas** og **Score**.



Efter man har oprettet **Score**, skal det konfigureres i **Inspector**'en.

Ud for **Font**, tryk på  og vælg **PressStart2P-Regular**. Sæt **Font Style** til **Bold** og **Font Size** til 25. Vælg en god farve i **Color**.

Kik nu i **Game** vinduet og se at teksten står helt nede i venstre hjørne. Det er lidt mærkeligt (men sådan er det bare). Flyt nu teksten op i toppen af skærmen (**Scene** vinduet), ved at trække den op. Når man zoom ud kan man se at der kommer en hvid firkant frem – det skal stå i toppen af denne firkant. Placer nu teksten et godt sted og sæt størrelsen op så den er lettere at se. Når man gør størrelsen større, så kan teksten godt pludselig "forsvinde". Det skyldes at den boks som teksten står i er for lille. Det rettes ved at vælge teksten og så rette boksens størrelse i **Scene**'en. Husk samtidig at lave boksen lang nok til den tekst som vi senere skal skrive.

Ret også **Text** til "Score: 0".

I **PlayerController** scriptet tilføj nu følgende linjer 'de rigtige steder i koden'.

```
using UnityEngine.UI;
...
[SerializeField] Text score;
...
score.text = "Score: "
```

Den sidste linje skal stå i bunden af **Update()** funktionen.

Træk **Score** objektet over i **Inspector**'en: **Player Controller->Score** feltet.

Kør koden og se at det virker.

3.8.1 Point

Lige nu vises der ingen point, da vi ingen point tæller!

Lav nu en variabel som hedder *point* af type *int*. Tæl den op hver gang man tager en frugt. Man viser pointene ved at opdatere linjen med **Score** teksten til følgende:

```
score.text = "Score: " + point;
```

Prøv og se hvordan det virker.

Hvis man ønsker at de forskellige frugter skal give forskellige point, så kan det gøres på følgende måde:

```
if (collision.gameObject.name == "Cherries")
    point += 20;
```

Her er det selvfølgelig vigtigt at alle *Cherries* hedder det samme, så man skal lige rette navnene i **Hierarchy**'et.

3.9 Simple forhindringer

Indtil nu har spillet været ret fredeligt, så det er på tide at tilføje nogle forhindringer. Der ligger en liste af forhindringer og fælder under **Traps** folderen. Under **Spikes** vælg *Idle* og træk den ind i **Scenen** et godt sted. I **Hierarchy**'et rename den til *Pigge*. Vælg *Pigge*, tilføj en **Collider** og tilpas størrelsen og sæt **Is Trigger**. Tilføj også et **Tag** som hedder *Trap*.

Tilføj nu den manglende kode i **PlayerController** scriptet, i funktionen `private void OnTriggerEnter2D(Collider2D collision)`

Husk en god lyd, når vi dør. Man er nødt til at forsinke ens død lidt, så man kan nå at afspille en lyd. Det gøres ved at tilføje en ekstra parameter til `Destroy` funktion, som angiver hvor mange sekunder man skal vente.

```
Destroy(gameObject, 1);
```

Put flere forskellige 'forhindringer' ind, som saven (**Saw**), og søg for at den er animeret (så det ser ud som om den kører rundt).

Husk at lave **PreFabs** af de forskellige forhindringer, så de er lette at genbruge.

3.9.1 Genstart spil

Når vi er døde, skal vi have en måde at genstarte spillet på.

Tilføj følgende linjer til koden:

```
using UnityEngine.SceneManagement;
...

StartCoroutine(RestartDelay());

...
private IEnumerator RestartDelay()
{
    yield return new WaitForSeconds(1);
    RestartLevel();
}

private void RestartLevel()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    Debug.Log("Restart");
}
```

Anden linje 'StartCoroutine ...' skal stå der hvor vi dør. Det er vigtigt at man fjerner

`Destroy(gameObject);` da al koden i scriptet ellers stopper!

Når man dør skulle spillet gerne starte forfra efter 1 sekund.

3.10 Bevægelige forhindringer

3.11 Udvid banen til flere 'skærme'

Vi begynder at løbe tør for plads i vores spil, så det er på tide at gøre banen bredere. Det kræver at kameraet følger *Player*, så vi stadig kan se hvad der foregår.

Åben *CameraFollower* scriptet og rename `CameraController` til `CameraFollower`. (Er muligvis allerede rettet).

Træk dette script op over *Main Camera* i **Hierarchy**'et. Vælg *Main Camera* og træk *Player* objektet over i **Player** i **Inspector**'en.

Nu skulle kameraet gerne følge *Player* og man kan derfor udvide sin bane i bredden.

3.12 Avanceret animationer

Lav en animation når man tager en frugt.

Hop animation

TBD Mangler beskrivelse ...

3.13 Tænd/sluk forhindringer

3.14 Flere baner

4 Unity Reference

Denne del vil nok enten blive skrevet om eller slettet i fremtiden, når al information fremgår af tutorial'en.

Dette er en lille Unity C# (C sharp) reference som beskriver nogle af de kodestumper man får brug for. Hvis man har brug for en guide til basal C#, så er der en god link i bunden af dette dokument.

Generelt er al C# kode markeret med denne font, og alle referencer til Unity GUI'en markeret med denne font.

4.1 Assets foldere

Det er en god ide at organisere alle sine filer undervejs, da det ellers kan være svært at finde rundt i dem senere. Det er derfor en god ide at starte med at oprette følgende foldere under **Assets**:

- *Scripts* (til alle C# scripts)
- *Animator* (til alle animationerne)
- *PreFabs* (til alle PreFabs)

Der vil blive behov for flere foldere efterhånden som man udvikler sit spil. Der ud over vil de assets man henter i *Asset store* automatisk komme i separate foldere.

4.2 Eksterne felter

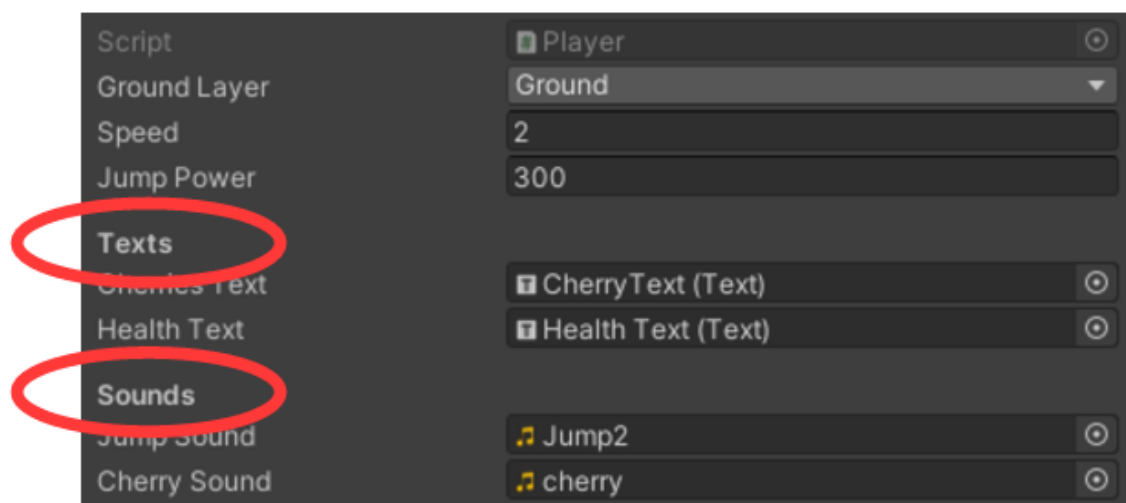
I mange tilfælde har man brug for at kunne se eller tilgå script variable fra Unity GUI'en. Ved at skrive `[SerializeField]` foran en variabel, bliver den synlig i **Inspector**'en.

```
[SerializeField] Text xxText;  
[SerializeField] GameObject[] xxList;
```

Den første linje laver et enkelt felt, mens den anden laver en liste af felter.

Hvis man har mange forskellige felter kan man indsætte nogle kosmetiske tekster for at gøre det mere overskueligt i **Inspector**'en.

```
[Header("Texts")] // Laver en tekst boks  
[Space]           // Laver en tom linje (afstand)  
[Tooltip("tip")]  // Laver et tooltip for næste linje, hvis man  
                  holder musen over feltet
```



Her er **Texts** og **Sounds** eksempler på `Header` felter.

4.3 Komponenter

Komponenter er alle de 'egenskaber' som man har tilknyttet til sit objekt, som *Sprite Render*, *Colliders*, etc. Man har tit brug for at kunne tilgå disse komponenter og de kan 'hentes' på følgende måde:

```
xxComp = GetComponent<Rigidbody2D>();
```

De enkelte komponentnavne kan findes i **Inspector**'en, hvor man bare skal fjerne eventuelle mellemrum fra navnet. Fx Rigidbody 2D -> Rigidbody2D

A screenshot of the Unity Inspector window showing the 'Rigidbody 2D' component selected. The component name is highlighted in blue.

Komponenten **Transform** er der altid og tilgås direkte med `transform`.

Hvis man her flere af den samme komponent, så skal man bruge:

```
xxCompList = GetComponent<CapsuleCollider2D>();
```

Funktionen har fået et 's' på og xxCompList er en liste af komponenter.

4.4 Taster

Når man skal aflæse hvilke taster der er trykket på, kan man med fordel bruge `GetAxisRaw()`, som til forskel fra en specifik tast er en samling af taster, som er mere generiske. Fx hvis man skal flytte en figur til siderne, så vil både tasterne A, D, venstre -og højre pil virke samt et eventuelt joystick (`Horizontal`). De enkelte kombinationer kan findes i **Edit->Projekt setting->Input manager**.

```
float dir;  
dir = Input.GetAxisRaw("Horizontal"); [-1, 0, 1]
```

For hop skal man bruge denne funktion, da den kræver at man slipper tasten før man kan hoppe igen.

```
float jumpKey;  
jumpKey = Input.GetButtonDown("Jump");
```

4.5 Bevægelse

Når man har fundet ud af hvilken flytning af figuren man har brug for, så laves selve flytningen på følgende måde:

```
rb = GetComponent<Rigidbody2D>();  
rb.velocity = new Vector2(xSpeed, ySpeed); // Move  
rb.AddForce(new Vector2(xForce, yForce)); // Jump
```

Hvis man har brug for at spejlvende sin figur, så den kigger i den rigtige retning, kan det gøres sådan:

```
sprite = GetComponent<SpriteRenderer>();  
sprite.flipX = true; // Flip image
```

Et fuldt eksempel kunne se sådan ud:

```

[SerializeField] float speed;

void Start()
{
    rb = GetComponent<Rigidbody2D>();
    sprite = GetComponent<SpriteRenderer>();
}

void Move(float dir)
{
    float xSpeed = dir * speed * 100 * Time.fixedDeltaTime;
    rb.velocity = new Vector2(xSpeed, rb.velocity.y);
    if (dir > 0)
        sprite.flipX = false;
    else if (dir < 0)
        sprite.flipX = true;
}

```

Ved bevægelse kan der også bruges `transform.Translate` (i stedet for `rb.velocity`):

```
transform.Translate(xSpeed, ySpeed, 0);
```

4.5.1 Hop

Inden man hopper er det vigtigt at man checker om man står på 'jorden', da man ellers kan hoppe i luften! Her er vist en af de måder det kan løses på. Man flytter sin 'kollisions kasse' lidt ned og ser om den overlapper med jorden.

```

[SerializeField] LayerMask groundLayer;

void Update()
{
    if (jumpKey && GroundCheck())
        Jump();
}

bool GroundCheck()
{
    coll = GetComponent<CapsuleCollider2D>();
    return Physics2D.CapsuleCast(coll.bounds.center,
        coll.bounds.size, 0f, 0f, Vector2.down, 0.1f, groundLayer);
}

```

Ground Layer skal sættes i **Inspector**'en, til det lag som er *Ground*.

Her er brugt en *CapsuleCollider*, men det virker med andre kolliders også.

Tallet **0.1f** angiver hvor meget kollideren flyttes ned. Man kan trimme lidt på det tal for at få den bedste effekt.

4.6 Kollision

Når to objekter, som begge har en **Collider** blok, rører hinanden, så har man en kollision. En kollision kan enten være en 'normal' kollision eller en 'trigger' kollision. En 'normal' kollision vil forhindre at de to objekter overlapper hinanden, fx som en person som går på et gulv, hvor en 'trigger' kollision blot er en indikation at to objekter overlapper, som hvis en person skal tage en ting. Hvis den ene af de to objekter har 'triggeren' sat, så bliver det en 'trigger' kollision. 'Triggeren' sættes i **Inspector**'en under **Collider**'en.

Is Trigger ☒

I scriptet kan man så lave en aktion på kollisionen, på følgende måde:

Normal kollision:

```
private void OnCollisionEnter2D (Collision2D collision)
{
    if (collision.gameObject.name == "Player")
    {
        ...
    }
}
```

Trigger kollision:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Cherry"))
    {
        ...
    }
}
```

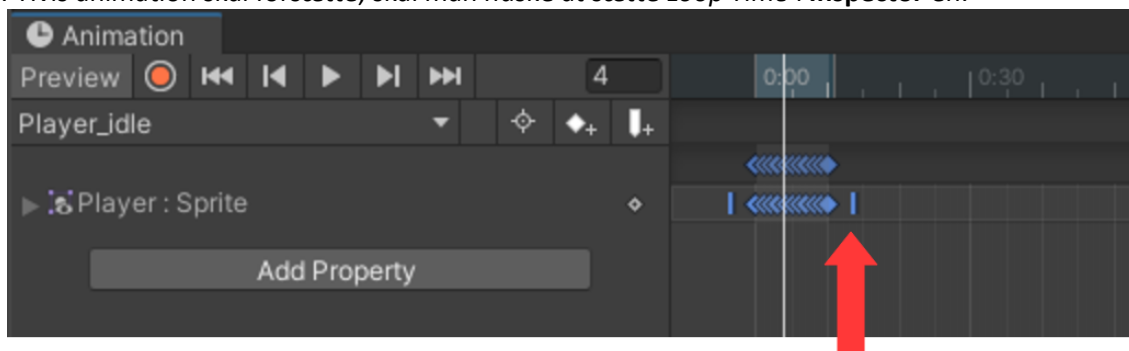
For at det virker skal mindst det ene objekt også have en *Rigidbody* komponent. Læg mærke til at der er vist to forskellige måde at 'se' hvilket objekt som man rammer – det er *ikke* afhængig af trigger metoden, men kun vist for at der er lidt at vælge imellem.

4.7 Animation

Når man skal have sin figur og objekters grafik til at bevæge sig skal man bruge en Animator. Dette er beskrevet i denne video: <https://youtu.be/GChUpPnOSkg?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U&t=14>

Her er hvordan man laver sin første animation til sin *Player* (kræver man har set videoen...). Hvis man ikke har en folder som hedder *Animator* under *Assets*, skal den laves først.

8. I folderen *Animator*, **Create->Animation** og kald filen noget 'fornuftigt', fx *Player_idle*.
9. Træk derefter *Player_idle* filen til objekt *Player* i **Hierarchy**'et. (Dette laver også en *Player controller* første gang man gør det.)
10. Så skal man åbne animationsvinduet: **Window->Animation->Animation**.
11. Vælg derefter *Player* objektet i **Hierarchy**'et.
12. Vælg nu alle de billeder som udgør animation og træk dem over i **Animatoren**.
13. Man kan nu se hvordan det ser ud ved at trykke på *play*, og justere hastigheden ved at flytte den blå streg til siderne (se billede nedenfor).
14. Hvis animation skal fortsætte, skal man huske at sætte *Loop Time* i **Inspector**'en.



Når man er færdig med animationsvinduet, kan man med fordel dock'et det til et af de andre vinduer man har, så det er lettere at finde næste gang.

4.7.1 Flere animationer (avanceret)

Hvis man har brug for flere forskellige animation til den samme figur, som stå og løbe, så gentager man skridtene oven for hver animation.

Nu skal man tilføje nogle transitioner og parametre til at styre det med (se video). Disse parametre kan så tilgås fra scriptet på følgende måde:

```
animator = GetComponent<Animator>();
animator.SetTrigger("die");
animator.SetFloat("velocity", speed);
```

Husk, at parametrene skal staves på helt samme måde som i **Animator**'en (die/velocity)!

4.8 Lyd

For at få lyd i sine spil, skal man først finde nogle lydfiler og lægge dem i **Assets/Sounds/**. For det *objekt* som man ønsker lydsupport for, skal man først tilføje en **AudioSource** komponent. I scriptet for det *objekt* gør man følgende:

```
[SerializeField] AudioClip jumpSound;

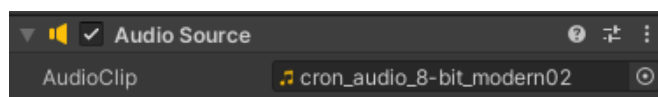
void Start()
{
    audioPlayer = GetComponent<AudioSource>();
}

void Update()
{
    if (jump)
        audioPlayer.PlayOneShot(jumpSound, 1.0f);
}
```

Man skal så trække den lydfil over i **Jump Sound** feltet i **Inspector**'en, som man ønsker at bruge.

4.8.1 Musik

Hvis man ønsker baggrundsmusik, skal man for **Main Camera** objektet tilføje en **AudioSource** komponent og så trække en lydfil over i **AudioClip** feltet og sætte **Loop** flaget.



4.9 Game tekst

Man indsætter en tekst på skærmen i spillet ved at lave et objekt med **UI->Text**. Det vil automatisk lave to objekter: **Canvas** og **objektet**. Dette objekt skal så 'linkes' til scriptet i **score** feltet – se nedenfor.



Efter man har oprettet objektet, kan det konfigureres i **Inspector**'en. Når man gør størrelsen større, så kan teksten godt pludselig "forsvinde". Det skyldes at den boks som teksten står i er for lille. Det rettes ved at vælge teksten og så rettet boksen størrelse i **Scene**'en.


```
using UnityEngine.UI;

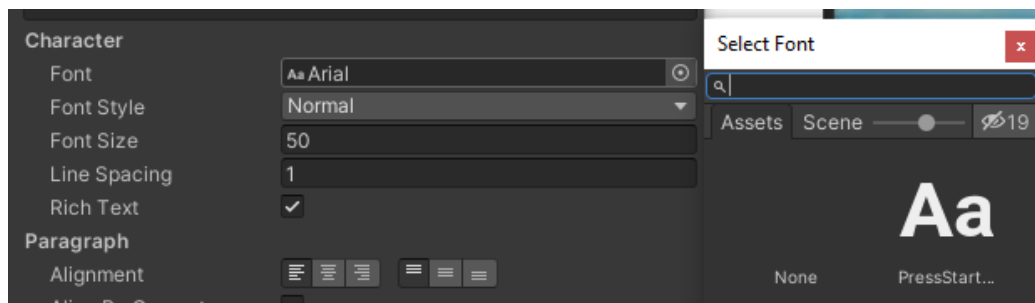
[SerializeField] Text score;
score.text = "Score: " + point;
```

Link: <https://youtu.be/pXn9icmreXE?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U&t=966>

4.9.1 Fonte

Default har Unity kun én font. Man kan let installere flere fonte og på den måde få et federe udseende. Font'ene kan bla. findes på den link, der er nederst i dette dokument.

Opret en folder under **Assets** og kald den **Fonts**. De fonte som man ønsker at bruge skal så trækkes over i denne folder. Når det er gjort, kan fonten bruges i tekst objektet ved at trykke på  ud for **Font**.



4.10 Prefabs

Prefabs er kopier af hele objekter (skabeloner), som så kan indsættes igen og igen. Det er den måde man kan kopiere et objekt mange gange i det samme spil som fx fjender og mønter.

De laves ved med musen at trykke på det ønskede objekt og så trække det ned i **Prefabs** folderen under **Assets**. (Hvis folderen ikke findes, skal den oprettes først). Derefter kan de indsættes i spillet med at trække dem fra **Prefabs** folderen ind i spillet (**Scene**).

4.11 Partikler

Partikler sættes op i GUI'en og kan så styres fra et script på følgende måde.

```
[SerializeField] ParticleSystem explosionParticle;  
  
explosionParticle.Play();  
explosionParticle.Stop();
```

4.12 Dynamisk opret og slet et objekt

Hvis man dynamisk vil oprette et objekt, fx til et skud, så skal man gøre følgende. Lav en Prefab af objektet i GUI'en.

I scriptet laves en variabel til at holde den Prefab som man vil kreere.

```
[SerializeField] GameObject objPrefab;
```

Man skal så bagefter ind i GUI'en og 'trække' den Prefab man vil bruge ind i det flet som er kommet i scriptet (**objPrefab**).

Derefter kan Prefab'en genereres på følgende måde:

```
Instantiate(objPrefab, pos, objPrefab.transform.rotation);
```

Hvor **pos** er en vektor som angiver, hvor objektet skal genereres.

Hvis et dynamisk objekt 'forsvinder' ud af skærmen eller på anden måde ikke er aktivt mere, er det vigtigt at det slettes igen, da der ellers over tid vil komme alt for mange 'døde' objekter. Man laver et script som skal sidde fast på det objekt som skal uddø. Scriptet kan se således ud – her undersøger man om objektet er uden for området, og i givet fald så slettes det.

```

void Update()
{
    // Destroy objekt if x position less than left limit
    if (transform.position.x < leftLimit)
    {
        Destroy(gameObject);
    }
}

```

4.13 Script kommunikation

Hvis man har brug for at et script kan 'se' en tilstand eller variable i et andet script, så kan det gøres på følgende måde.

I dette eksempel, så findes der et objekt som hedder '**Player**' som har et script som hedder **PlayerController** og dette script har en *public* variabel som hedder `gameOver`;

Hvis man vil tilgå denne `gameOver` variabel fra et andet script, skal man gøre følgende:

```

playerController playerControllerScript;

void Start()
{
    playerControllerScript =
        GameObject.Find("Player").GetComponent<PlayerController>();
}

void Update()
{
    if (playerControllerScript.gameOver == false)
        <gør noget>
}

```

5 Utility

5.1 Tidshåndtering

For hver frame i spillet, er der gået tid siden sidste frame. Denne tid er god at bruge når man i hver loop fx bevæger en figur. Tiden er i sekunder (dvs. et kommatall):

```
Time.deltaTime;
```

Så hvis man skal lave en hastighed, som er uafhængig af frameraten, så kan det gøres på følgende måde:

```
move = speed * Time.deltaTime;
```

hvor `speed` angiver en faktor for hvor hurtigt noget skal flytte sig.

Hvad man skal lave et delay mellem fx to skud, kan man gøre følgende:

```

tidVent += Time.deltaTime;
if (tidVent > 2) { // 2 sekunder
    <skud>
    tidVent = 0;
}

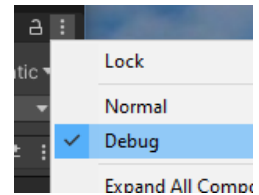
```

5.2 Fejlsøgning

Når man skal fejlsøge sine C# scripts, kan man med fordel sætte nogle udskrifter ind i kode for at se hvordan forskellige værdier ændre sig. Man kan sætte en debug tekst ind på følgende måde, som kan ses i **Console** når programmet kører.

```
Debug.Log("Text " + value);
```

Man kan også i GUI'en **Inspector** trykke på de tre prikker i højre hjørne og vælge **Debug**. På den måde kan man i **Inspector** under **Script** se alle variable i programmet mens det kører.



5.3 Interval check

Hvis man har en værdi som skal holde sig inde i et interval, fx antal liv, så kan man bruge en **Clamp** funktion.

```
newLevel = Mathf.Clamp(level - 1, 0, 3);
```

Første parameter er den nye værdi som skal testes, næste er min og derefter max. Funktionen returnerer en lovlig værdi så tæt på den nye værdi som muligt.

5.4 Forsinkelse

Nogle gange har man brug for at lave en forsinkelse mellem to operationer. Dette kræver at man laver en ny funktion til dette og at den kaldes på en 'speciel' måde.

```
StartCoroutine(xxDelay());

IEnumerator xxDelay()
{
    <lav noget>
    yield return new WaitForSeconds(2); // Vent 2 sekunder
    <lav noget>
}
```

Hvor **xxDelay** er navnet på den nye funktion.

5.5 Importer Unity pakke

Importer en Unity pakke (package).

I GUI'en vælg: **Assets->Import Package->Custom package** og vælg filen. I det vindue som kommer frem trykkes på **Import** i nederst højre hjørne. Pakken er nu importeret og ligger under **Assets**.

6 Specielle scripts

6.1 Bevægelige objekter

Her er et script som kan få et objekt til at bevæge sig i mellem en serie af punkter. Det kan bruges til at lave en bevægelig platform, men også fjender som bevæger sig i et mønster. Ideen er beskrevet i denne video:

Link: <https://youtu.be/UIEE6wjWuCY?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U&t=198>

Scriptet oprettes i **Assets/Scripts/** og følgende indhold tilføjes: (Scriptet er en del af **cpf_start_pakke**)
WaypointFollower.cs

```

[SerializeField] GameObject[] waypoints;
int currentWaypointIndex = 0;
[SerializeField] float speed;
SpriteRenderer sr;

private void Start()
{
    sr = GetComponent<SpriteRenderer>();
}

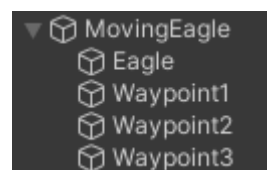
void Update()
{
    int prevWaypointIndex;

    if
(Vector2.Distance(waypoints[currentWaypointIndex].transform.position,
transform.position) < 0.1f)
    {
        prevWaypointIndex = currentWaypointIndex;
        currentWaypointIndex++;
        if (currentWaypointIndex >= waypoints.Length)
            currentWaypointIndex = 0;
        // Change direction for default layer only
        if (gameObject.layer != LayerMask.NameToLayer("Ground"))
        {
            if
(waypoints[currentWaypointIndex].transform.position.x >
waypoints[prevWaypointIndex].transform.position.x)
                sr.flipX = true;
            else
                sr.flipX = false;
        }
        transform.position = Vector2.MoveTowards(transform.position,
waypoints[currentWaypointIndex].transform.position, Time.deltaTime *
speed);
    }
}

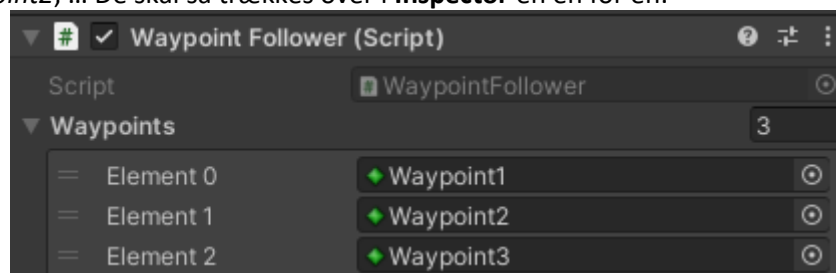
```

Scriptet vender automatisk objektet så det peger i "den rigtige retning", på nær hvis det er **Ground** layer. På den måde kan scriptet bruges både til fjender som bevæger sig og platforme som skal bevæge sig.

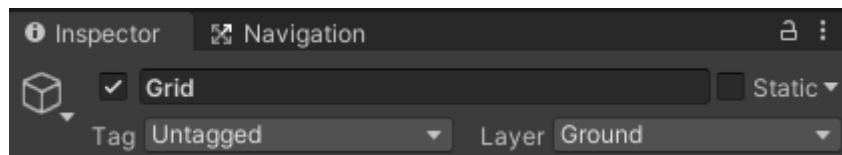
Det kan være en fordel at lave en objektgruppe i **Hierarchy**'et, så objektet og waypoints er samlet.



Man tilføjer scriptet til det objekt som skal bevæge sig. I **Inspector**'en laves der det antal Waypoints som man har brug for – typisk 2. Dernæst laves et tilsvarende antal Waypoints i **Hierarchy**'et og de kaldes *Waypoint1*, *Waypoint2*, ... De skal så trækkes over i **Inspector**'en en for en.



Hvis det er en platform, som skal bevæge sig er det vigtigt at man har sat **Layer** til *Ground*. Hvis det ikke findes, skal det bare oprettes.



6.2 Stå fast på platforme som bevæger sig

Hvis man står på en platform der bevæger sig, så er det vigtigt at figuren bevæger sig sammen med platformen for ellers falder figuren af. Scriptet oprettes i **Assets/Scripts/** og tilføjes til de platforme, som skal have denne egenskab. Det er vigtigt at ens figur hedder *Player*, ellers kan man bare rette det i scriptet. (Scriptet er en del af *cpf_start_pakke*)

StickyPlatform.cs

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.name == "Player")
    {
        collision.gameObject.transform.SetParent(transform);
    }
}
private void OnCollisionExit2D(Collision2D collision)
{
    if (collision.gameObject.name == "Player")
    {
        collision.gameObject.transform.SetParent(null);
    }
}
```

7 Special setup

7.1 Flyt kamera efter spiller

Hvis man gerne vil have kameraet til at følge figuren, så skal man lave følgende script og tilføje det til **Main Camera** objektet. (Husk at sætte figuren ind i **player** feltet i **Inspector**'en.) (Scriptet er en del af *cpf_start_pakke*)

CameraFollower.cs

```
[SerializeField] Transform player;

void Update()
{
    transform.position = new Vector3(player.position.x,
        transform.position.y, transform.position.z);
}
```

Link: <https://youtu.be/PA5DgZfRsAM?list=PLgOEwFbvGm5o8hayFB6skAfa8Z-mw4dPV&t=321>

7.2 Glat overflade

Hvis man ikke vil kunne "hænge fast" på vægge og platformsider, så skal de have en glat overflade. Det gøres ved at lave en fysisk overflade under **Assets** folderen (**Create->2D->Physics Material 2D**) og sætte **Friction** til 0. Derefter vælges den i **Collider**'en under **Material** for den overflade man ønsker 'glat'.

7.3 Lyd support (avanceret)

Hvis man har brug for at lave lyd mange forskellige steder i sit spil, kan man oprette en central lyd manager. Det gør det let lettere for de enkelte scripts.

Tilføj et audio objekt og kald det **SoundManager**:



Lav en scriptfil (`SoundManager.cs`) og tilføj følgende indhold og tilføj den til **SoundManager** objektet.

```
public static SoundManager instance { get; private set; }
    AudioSource source;

    private void Awake()
    {
        instance = this;
        source = GetComponent();
    }

    public void PlaySound(AudioClip sound)
    {
        source.PlayOneShot(sound);
    }
```

Lydfiler lægges i **Assets/Sounds/**. Man kan nu afspille en lydfil fra et vilkårligt script på følgende måde (husk at selve lydfilen skal sættes ind i **xxSound** feltet i **Inspector**'en):

```
[SerializeField] AudioClip xxSound;

SoundManager.instance.PlaySound(xxSound);
```

Link: <https://www.youtube.com/watch?v=7e6GJtm3FU4&list=PLgOEwFbvGm5o8hayFB6skAfa8Z-mw4dPV&index=12>

8 Links

God start serie	https://www.youtube.com/playlist?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U
Skud eksempel	https://www.youtube.com/watch?v=PUPC44Q64zY&list=PLgOEwFbvGm5o8hayFB6skAfa8Z-mw4dPV&index=4
Tips videoer	TOP 10 UNITY TIPS - 2017 TOP 10 UNITY TIPS #2
Unity manual	https://docs.unity3d.com/2020.2/Documentation/ScriptReference/index.html
Unity assets store	https://assetstore.unity.com
Fonts	https://fonts.google.com/ (Fx 'Press Start 2D')
Dansk C#/Unity document	https://github.com/Grailas/CodingPiratesAalborg/blob/master/Guides/Hj%C3%A6lpegui de.pdf