

Unity Intro

Af: Michael Hansen, Coding Pirates Furesø, 2022, version 0.05

Dokumentet ligger her: <https://github.com/mhfalken/unity/>



Formålet med dette dokument er at gøre det lidt lettere at komme i gang med Unity og lidt lettere at huske hvordan man lige laver en speciel ting. Der findes masser af gode YouTube videoer, men desværre også mange som ikke gør det på den optimale måde og dermed gør det mere besværligt end nødvendigt og nogle gange decideret forkert.

Denne guide kan ikke stå alene, men er en god støtte mens man lærer Unity eller hvis man har brugt det lidt, men lige har glemt hvordan man gør.

Dokumentet indeholder i bunden nogle links til gode videoer og i nogle afsnit specifikke links til de tips som er beskrevet.

Dokumentet er primært lavet til **2D platform** spil.

1	Installation af Unity på PC/MAC.....	3
2	Microsoft Visual Studio editor.....	3
3	Unity C# Reference	4
3.1	Assets foldere	4
3.2	Eksterne felter	4
3.3	Komponenter.....	5
3.4	Taster	5
3.5	Bevægelse.....	5
3.6	Kollision	6
3.7	Animation	7
3.8	Lyd	8
3.9	Game tekst.....	8
3.10	PreFabs	9
3.11	Partikler	9
3.12	Dynamisk opret og slet et objekt.....	9
3.13	Script kommunikation	10
4	Utility	10
4.1	Tidshåndtering.....	10
4.2	Fejlsøgning.....	11
4.3	Interval check	11
4.4	Forsinkelse.....	11
4.5	Importer Unity pakke	11
5	Specielle scripts	12
5.1	Bevægelige objekter	12
5.2	Stå fast på platforme som bevæger sig.....	13
6	Special setup.....	13
6.1	Flyt kamera efter spiller.....	13
6.2	Glat overflade	14
6.3	Lyd support (avanceret)	14
7	Links	14

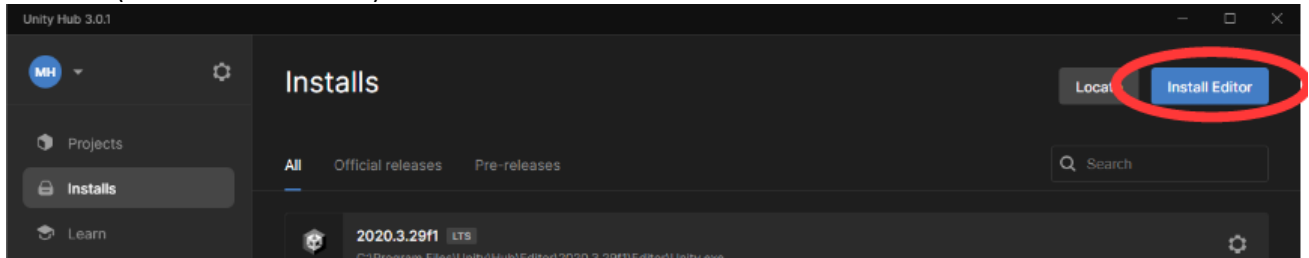
1 Installation af Unity på PC/MAC

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).

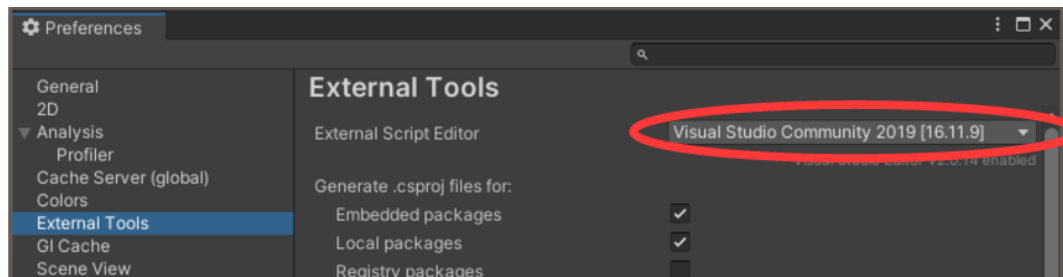


Her skal man vælge en *officiel release* fx 2020.3.xxxx (**LTS**). Den fylder ca. 6GB og det tager derfor noget tid at hente afhængig af internetforbindelse og computeren. Når den er downloadet, så installeres den automatisk. Det hele tager fra 10 minutter til nogle timer ...

Når den er færdig med installationen er man klar til at lave sit første projekt. Indenfor 30 dage skal man logge ind på sin Microsoft Visual Studio editor med en Microsoft konto/email, eller oprette en email til det, ellers løber demolicensen ud og MVS holder op med at virke.

Link: <https://youtu.be/li-scMenaOQ?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U&t=99>

Første gang man starter et Unity projekt, skal man gå ind i **Edit->Preferences->External tools** og checke at der står "Visual Studio Community 2019" i den røde cirkel (se billede). Hvis ikke så vælg den i dropdown menuen.



2 Microsoft Visual Studio editor

Den editor (det program som man skriver sin C# kode i) som følger med er Microsoft Visual Studio 2019 (MVS).

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 Unity C# Reference

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.**

3.1 Assets foldere

Det er en god ide at organisere alle sine filer undervejs, da det ellers kan være svært af 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.

3.2 Eksterne felter

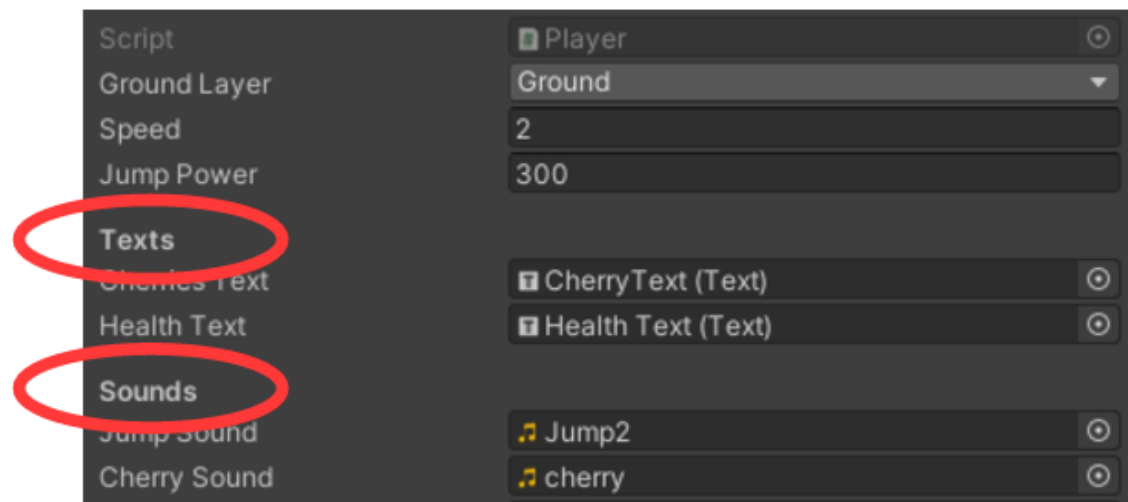
I mange tilfælde har man brug for at kunne se eller tilgå script variable fra Unity GUI'en. Ved at skrive `[SerializeField]` foran en variabel, bliver den synlig i **Inspector'en**.

```
[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
```



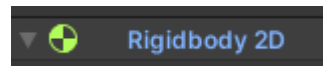
Her er **Texts** og **Sounds** eksempler på **Header** felter.

3.3 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



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

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

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

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

3.4 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");
```

3.5 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 kikker 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);
```

3.5.1 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.

3.6 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.



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"))
    {
        ...
    }
}
```

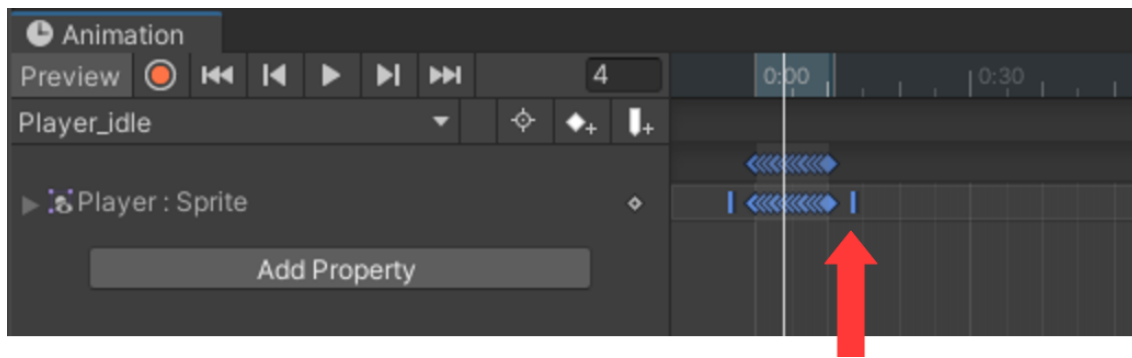
For at det virker skal mindst det ene objekt også have en *Rigidbody* komponent. Læg mærke til at der er vist to forskellige måde at 'se' hvilket objekt som man rammer – det er *ikke* afhængig af trigger metoden, men kun vist for at der er lidt at vælge imellem.

3.7 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/GChUpnOSkg?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.

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.)
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**.
6. 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).
7. Hvis animation skal forsætte, 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.

3.7.1 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)!

3.8 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;

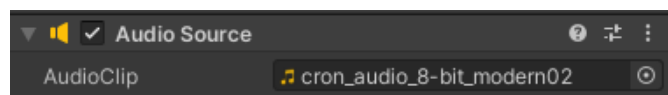
void Start()
{
    audioPlayer = GetComponent<AudioSource>();
}

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.

3.8.1 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.



3.9 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;

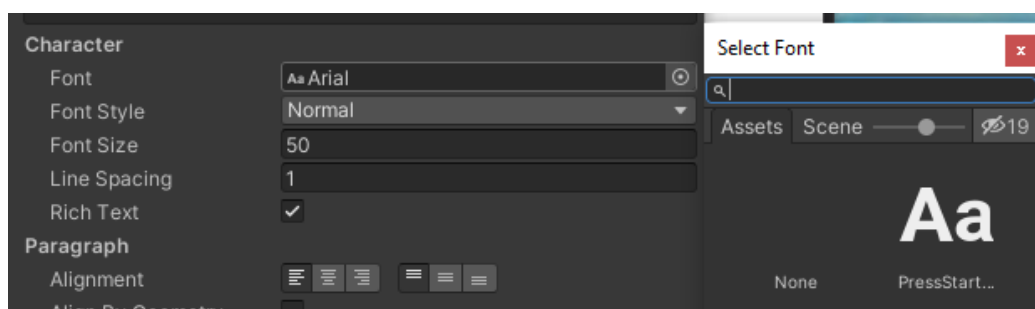
[SerializeField] Text score;
score.text = "Score: " + point;
```

Link: <https://youtu.be/pXn9icmreXE?list=PLrnPJChvNZuCVTz6lvhR81nnaf1a-b67U&t=966>

3.9.1 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**.



3.10 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**).

3.11 Partikler

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

```
[SerializeField] ParticleSystem explosionParticle;

explosionParticle.Play();
explosionParticle.Stop();
```

3.12 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);
    }
}
```

3.13 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>
}
```

4 Utility

4.1 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;
}

```

4.2 Fejlsøgning

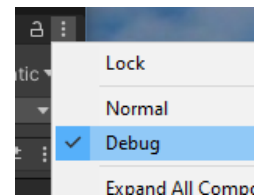
Når man skal fejlsøge sine C# scripts, kan man med fordel sætte nogle udskrifter ind i kode for at se hvordan forskellige værdier ændre sig. Man kan sætte en debug tekst ind på følgende måde, som kan ses i **Console** når programmet kører.

```

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.



4.3 Interval check

Hvis man har en værdi som skal holder 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.

4.4 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.

4.5 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**.

5 Specielle scripts

5.1 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/UlEE6wjWuCY?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U&t=198>

Scriptet oprettes i **Assets/Scripts/** og følgende indhold tilføjes:

WaypointFollower.cs

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

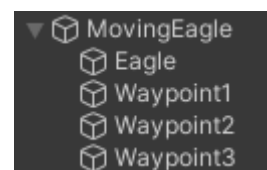
private void Start()
{
    sr = GetComponent<SpriteRenderer>();
}

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.

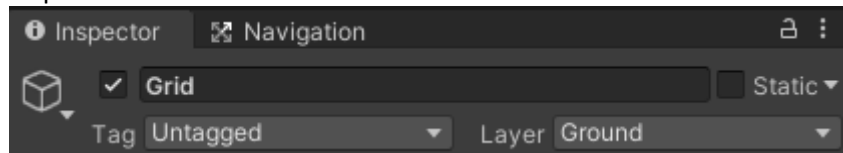
Det kan være en fordel at lave en objektgruppe i **Hierarchy**'et, så objektet og waypoints er samlet.



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.



5.2 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.

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);
    }
}
```

6 Special setup

6.1 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.)

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>

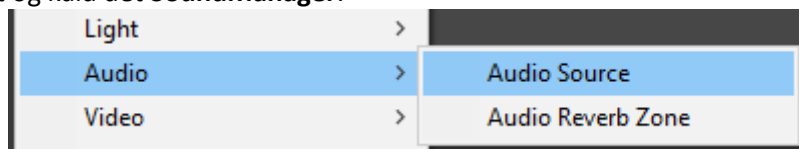
6.2 Glat overflade

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'.

6.3 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>

7 Links

God start serie	https://www.youtube.com/playlist?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U
Skud eksempel	https://www.youtube.com/watch?v=PUc44Q64zY&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%A6lpeguide.pdf
-------------------------	---