ImpactLab - Maximizing your CS Skills
Lecture 3: Our First Game

Summer 2022

School of Computing and Data Science

Wentworth Institute of Technology

Volcano Island

Back to Simple Scripts

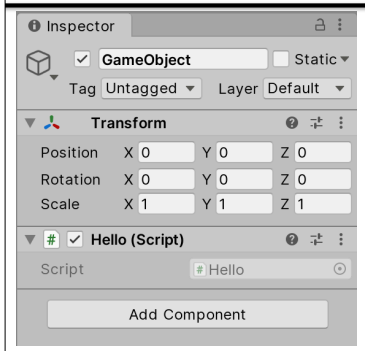
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hello : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Create a new script in Unity (Right Click in Project-> Create-> C# Script)

Create a new empty object in the Hierarchy. (Right Click-> Create Empty

Drag the script from the Project view to the empty you just created.

Simple Scripts



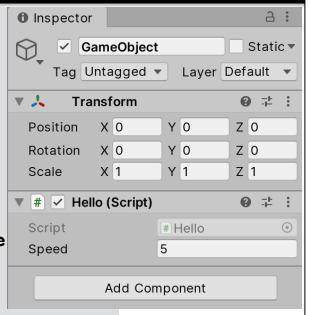
Notice that this gameobject (GameObject) has two components, **`Transform`** and **`Hello`** (my script).

Your scripts can access any component via code!

The other elements, Tag, Layer, etc., are adjustments that we can make to the gameobject itself, we'll talk about them in the future.

Simple Scripts

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hello : MonoBehaviour
{
    public float speed = 5f;
    private float health = 10f;

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

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```



Any public variable in a MonoBehavior (attached to an object) shows up in the inspector.

Simple Scripts

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hello : MonoBehaviour
{
    public float speed = 5f;
    private float health = 10f;

    // Start is called before the first frame update
    void Start()
    {
        Vector3 pos = this.transform.position;
        Debug.Log(pos);  //prints (0.0, 0.0, 0.0)
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

You can access the transform (and thus the position, rotation, and scale) directly using **`this.transform`**

Simple Scripts

Add a Rigidbody component to this object.

Other components need to be accessed via the **`GetComponent()`** call.

```
Type Name = GetComponent<Type>();
```

With this reference, you can access any element of the component via code.

Simple Scripts

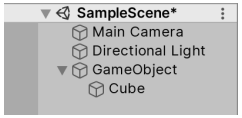
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hello : MonoBehaviour
{
    public float speed = 5f;
    private float health = 10f;
    private Rigidbody rb;

    // Start is called before the first frame update
    void Start()
    {
        Vector3 pos = this.transform.position;
        Debug.Log(pos);  //prints (0.0, 0.0, 0.0)
        rb = GetComponent<RigidBody>();
    }

    // Update is called once per frame
    void Update()
    {

    }
```

If the component doesn't exist, you will get an error on the **`GetComponent()`** line

There are workarounds for this that we'll discuss later.

Movement Example

- Add a Cube to the GameObject that we created. Right Click in the Hierarchy->3D Object->Cube and drag it onto the GameObject.




```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hello : MonoBeha
{

  Update()
  {

  }

}
```

Clean out the script that we've been working on, we'll only need **`Update()`** for now.

Also, remove the **`Rigidbody`** from the object.

Movement Example

Clean out the script that we've been working on, we'll only need **`Update()`** for now.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hello : MonoBehaviour
{

  public float speed = 10f;

  Update()
  {
    Vector3 pos = this.transform.position;
    if(Input.GetKey(KeyCode.UpArrow)){
      pos.y += speed * Time.deltaTime;
    }
    this.transform.position = pos;
  }
}
```

Grab the current position as **`pos`**

If the Up Arrow key is pressed...

...change **`pos.y`**

Update the **`transform.position`**

Movement Example

```
...
    Vector3 pos = this.transform.position;
    if(Input.GetKey(KeyCode.UpArrow)){
      pos.y += speed * Time.deltaTime;
    }
...
```

How is this updating the y position?

`speed` is a constant that we set

`pos.y` is being increased

What is **`Time.deltaTime`**?

`Time.deltaTime` is the amount of time that has passed between this frame and the previous frame.

For a 60fps game, this time is about 16ms

Movement Example

```
...
    Vector3 pos = this.transform.position;
    if(Input.GetKey(KeyCode.UpArrow)){
      pos.y += speed;
    }
...
```

What if we didn't have it? How does our movement behave?

Physics Lesson

$$\frac{dx}{dt} = v$$

Velocity is change in distance over change in time

$$\frac{x(t+dt) - x(t)}{dt} = v$$

Using definition of the derivative

$$x(t+dt) = v \cdot dt + x(t)$$

Rearrange

Movement Example

$$x(t+dt) = v \cdot dt + x(t)$$

Position Now

Position **`dt`** time from now

`dt` (small time)

`speed` (Velocity)

Look Familiar?

Unity has its own physics system (based on **`Rigidbody`**) that we'll use in the future, but at a fundamental level, it's based on numerically solving differential equations.

So, what is **`Time.deltaTime`**?

It's **`dt`** in our equation.

Which means we're using real physics to move our object.

Movement Example

```
...
  Update()
  {
    Vector3 pos = this.transform.position;
    if(Input.GetKey(KeyCode.UpArrow)){
      pos.y += speed * Time.deltaTime;
    }
    if(Input.GetKey(KeyCode.DownArrow)){
      pos.y -= speed * Time.deltaTime;
    }
    if(Input.GetKey(KeyCode.RightArrow)){
      pos.x += speed * Time.deltaTime;
    }
    if(Input.GetKey(KeyCode.LeftArrow)){
      pos.x -= speed * Time.deltaTime;
    }
    this.transform.position = pos;
  }

}
```

Add the rest of the directions, pay attention to the axis (**x** or **y**) and the sign (**-=** or **+=**).

Keep in mind:

This is moving the GameObject, not the cube.

But, since the cube is a child (in the Hierarchy) of the GameObject, it moves too.

Game Design Document (GDD)

- Every game has something that could be called a game design document, or bible, or Wiki, etc.
- This describes the details of game: Characters, Mechanics, Sound/Music, Theme/Genre, UI, Monetization, etc.
- I've created a GDD for Volcano Island as a simple example.

Let's look at the VI GDD now

Game Design Document (GDD)

GDDs are meant to keep your team focused and can allow new members to see the big picture before starting work.

Other GDDs:

Doom: https://5years.doomworld.com/doombible/doombible.pdf

Deus Ex: https://drive.google.com/file/d/0B2_-knUokw90RzIwUEJsU2pYdWs/view

It is very easy to add too many features or forget important notes about your game, the GDD will keep you organized.

Many companies use the GDD as a "living" document that changes throughout the course of development.

Game Design Document (GDD)

- Many times, there are also other supporting documents:
  - Technical Design Document (TDD): includes implementation details about the game like internal APIs and external libraries.
  - Artistic Design Document: lays out the artistic theme/genre for the artists in the group.
  - Concept Document: a smaller document that acts as the vision for the game and as a sales tool to give to potential publishers.

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - GameManager
  - MusicManager
  - Player
  - Enemy
  - Spawner
  - Menu

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - **GameManager**
  - MusicManager
  - Player
  - Enemy
  - Spawner
  - Menu

The **`GameManager`** is a *singleton* class that contains functionality to display/change health, the death display, switching scenes and reseting the game.

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - GameManager
  - **MusicManager**
  - Player
  - Enemy
  - Spawner
  - Menu

The **`MusicManager`** is a *singleton* class that plays the background music. It will persist between scene changes and must have an accessible **`AudioSource`** component.

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - GameManager
  - MusicManager
  - **Player**
  - Enemy
  - Spawner
  - Menu

The **`Player`** class contains all the information about the player character: speed, health, input, special effects, and animations. Must be attached to a game object with a **`Rigidbody2D`** and (later) an **`Animator`** component.

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - GameManager
  - MusicManager
  - Player
  - **Enemy**
  - Spawner
  - Menu

The **`Enemy`** class contains all the information about the falling fireballs. This includes speed, special effects, movement, position, and collision logic. Must be attached to a game object with a **`Collider`** component.

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - GameManager
  - MusicManager
  - Player
  - Enemy
  - **Spawner**
  - Menu

The **`Spawner`** class contains variables associated with the spawning of the fireballs. The includes random number generation, difficulty increases and the location of all possible spawn locations and currently spawned fireballs.

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - GameManager
  - MusicManager
  - Player
  - Enemy
  - Spawner
  - **Menu**

The **`Menu`** class is the smallest class in the game. It only contains the functionality to load the main game scene when the player clicks the player button on the main menu. This is the last class that we will create.

Volcano Island Technical Overview

- The game will consist of six scripts/classes:
  - GameManager
  - MusicManager
  - Player
  - Enemy
  - Spawner
  - Menu

All of these scripts will be attached to game objects (all derive from **`MonoBehaviour`**). Some have other component requirements because we'll make certain assumptions about what we can access in code.

What is a Design Pattern?

- Basically: A problem/solution pair.
  - A technique to repeat successful solutions to common problems that appear in applications.
- Borrowed from Civil and Electrical Engineering.
- Think of it as a structured approach to coding your applications that sits midway between the features of the language and a concrete algorithm.

Pattern Categories

- **Creational Patterns:** Deal with the process of creating objects from classes.
- **Structural Patterns**: Concerned with the integration and composition of classes and objects
- **Behavioral Patterns**: Deal with the communication between objects/classes

The Singleton

- Creational Pattern
- **Problem**: A class only ever needs one instantiated object that should be available globally. More will either interfere or don't make sense.
- **Solution**: Only allow one instantiation, via a private constructor, and allow a single global access point to reference the single instance.

This is the typical idea behind the singleton, different languages may have sightly different implementations.

Simple Singleton in Unity

```
public class GameManager : MonoBehaviour
{
    private static GameManager _instance = null;

    void Awake(){
        if(_instance == null){
            _instance = this;
        }
    }

    public static GameManager instance(){
        return _instance;
    }
}
```

`Awake()` happens before **`Start()`**

Since our script will be attached to a gameobject, the class gets instantiated when the gameobject gets created.

The Singleton in Unity

```
public class GameManager : MonoBehaviour
{
    private static GameManager _instance = null;

    void Awake(){
        if(_instance == null){
            _instance = this;
        }
    }

    public static GameManager instance(){
        return _instance;
    }
}
```

Now, how do you access the **GameManager?**

```
GameManager gm = GameManager.instance();
```

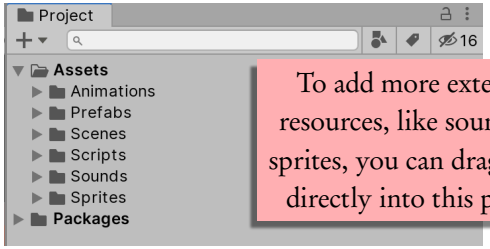

Volcano Island

- We're going to start with a very simple game to get used to the terminology and using Unity.
- I've included a starter repo for you, it includes basic some organization and sprite graphics.

All of these folders will have files in them by the time we are done.

Volcano Island

- We're going to start with a very simple game to get used to the terminology and using Unity.
- I've included a starter repo for you, it includes basic some organization and sprite graphics.

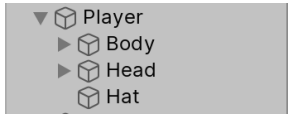


To add more external resources, like sounds or sprites, you can drag them directly into this panel.

f these folders will iles in them by the ne we are done.

Volcano Island: Starting Player

- Create a new empty object in the Hierarchy and call it "Player".
- In the Sprite folder, drag each Snowman sprite (Head, Body, Hat) into the Player gameobject.



Each part of the snowman is separate (we'll animate them later). Because they are part of the "Player" object, we can move the pieces *relative* to the position of the "Player" gameobject (set to 0,0,0)

Most of the time, we'll create an empty gameobject and drag graphics, scripts, etc. onto it.

Volcano Island: Player Script

- We need to create a script that allows for basic movement.
- For now, we need values for **speed**, **input**, and **health**.
- The actual movement will be handled by the Unity physics system, so we need a **Rigidbody2D**.
- Later, we'll add animations, particle effects, functionality to take damage and reset the player.
- The player script will be attached to the "Player" object, not any of the specific sprites.

Volcano Island: Player Script

```
[RequireComponent(typeof(Rigidbody2D))]
public class Player : MonoBehaviour
{
  public float speed;
  private float input;
  public int health;
  private float eps = 0.0001f;
  Rigidbody2D rb;

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

  void FixedUpdate()
  {
    input = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(input*speed, rb.velocity.y);
  }
```


Volcano Island: Player Script

```
[RequireComponent(typeof(Rigidbody2D))]
public class Player : MonoBehaviour
{
  public float speed;
  private float input;
  public int health;
  private float eps = 0.0001f;
  Rigidbody2D rb;

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

  void FixedUpdate()
  {
    input = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(input*speed, rb.velocity.y);
  }
```

RequireComponent: The object that this script is attached to must have the **typeof** component

Volcano Island: Player Script

```
[RequireComponent(typeof(Rigidbody2D))]
public class Player : MonoBehaviou
{
  public float speed;
  private float input;
  public int health;
  private float eps = 0.0001f;
  Rigidbody2D rb;

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

  void FixedUpdate()
  {
    input = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(input*speed, rb.velocity.y);
  }
```

There is our **speed**, **health**, **input**, **eps** and a reference to the **Rigidbody2d**.

speed and **health** are public, so they are shown in the inspector (*make sure to set them to a value!*).

eps (epsilon) is used to help with input in the future.

Volcano Island: Player Script

```
[RequireComponent(typeof(Rigidbody2D))]
public class Player : MonoBehaviour
{
  public float speed;
  private float input;
  public int health;
 Rigidbody2D rb;

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

  void FixedUpdate()
  {
    input = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(input*speed, rb.velocity.y);
  }
```

For now, **`Start()`** only needs to get a reference to the **`Rigidbody2D`** that must be attached to this object.

Volcano Island: Player Script

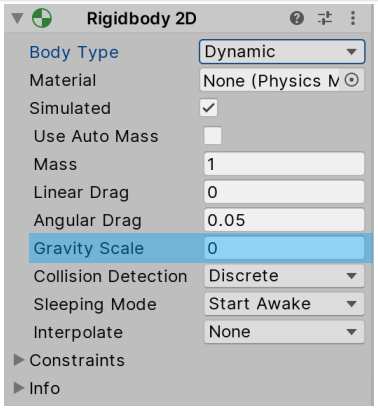
```
[RequireComponent(typeof(Rigi
public class Player : MonoBeh
{
  public float speed;
  private float input;
  public int health;
  private float eps = 0.0001f
  Rigidbody2D rb;

  void Start()
  {
    rb = GetComponent<Rigidbo
  }

  void FixedUpdate()
  {
    input = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(input*speed, rb.velocity.y);
  }
```

`FixedUpdate()` only runs when the physics system updates. Any interaction with the physics system (the **`Rigidbody2D`**), should be done in **`FixedUpdate()`**.

Here, we add a velocity to the y component of the Rigidbody. This value is the input value multiplied by the speed.

Volcano Island: Player Script

- We also need to configure the Rigidbody2D before it will work correctly for us.
- Adding the Rigidbody immediately turns it into a physics object, so it will be affected by forces like gravity.
- For your snowman, it will only move left and right, so we don't need gravity at all.



There's a lot here that we'll leave default

Volcano Island: Enemy

- At a fundamental level, our snowman will have to avoid fireballs will rain down from the sky.
- For now, the enemy script will be very simple, later we will add particle effects and collision code.
- Create a new empty gameobject called "Enemy" and a script called "Enemy".
- Add the script to the "Enemy" gameobject.

Volcano Island: Enemy Prefab

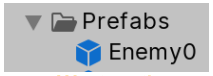
- Add one of the fireball sprites as a child of the "Enemy" gameobject (similar to the snowman).

We are going to create a **Prefab** from this object.

Creating a **Prefab** in Unity is a way to store a gameobject with all of its setting/components intact.

We will then be able to instantiate this prefab through code whenever we need.

Drag the "Enemy" object, in the Hierarchy, directly into the Prefabs folder in the project tab.



Volcano Island: Enemy Script

```
public class Enemy : MonoBehaviour
{
    public int damage;

    //two speeds to pick a random value between
    public float minSpeed;
    public float maxSpeed;

    private float speed;

   void Start()
    {
        //pick random value between min and max speed
        speed = Random.Range(minSpeed, maxSpeed);
    }

    void Update()
    {
        //translate this object down at speed
        this.transform.Translate(Vector3.down * speed * Time.deltaTime);
    }
```


Volcano Island: Enemy Script

```
public class Enemy : MonoBehaviour
{
    public int damage;

    //two speeds to pick a random value bet
    public float minSpeed;
    public float maxSpeed;

    private float speed;

   void Start()
    {
        //pick random value between min and
        speed = Random.Range(minSpeed, maxS
    }

    void Update()
    {
        //translate this object down at speed
        this.transform.Translate(Vector3.down * speed * Time.deltaTime);
    }
```

We'll randomly pick a speed for the fireball to fall when the object is instantiated.

We are *not* using the physics system, we are just moving the object down using the **`Translate()`** function.

This is just a different way to move objects without using Rigidbodies.

Later, we'll add a collider to determine when we've hit the player and more code to instantiate for particle effects

Volcano Island: Intermission

- We should have a moveable snowman and fireballs that fall from the sky.
- Currently, the fireballs never despawn and there is no collision with the snowman.
- Upcoming: collision (with ground and snowman), background sprites, GameManager.

Volcano Island: Spawner Script

```
public class Spawner : MonoBehaviour
{
    public GameObject[] enemies;

    //public variables to control spawn behavior
    public float timeBetweenSpawns;
    public float minSpawnTime;
    public float decreaseAmt;
    private float spawnTimer;

    void Start()
    {
        spawnTimer = 0f;
    }
```

Again, create a new object, and attach the Spawner script to it.

Here we have a few variables that control the spawn delay and the "difficulty".

Also, we have an array of enemy objects that we'll randomly pick from when spawning.

Volcano Island: Spawner Script

```
    // Update is called once per frame
    void Update()
    {
        if (spawnTimer <= 0) {
            //pick random enemy prefab
            GameObject enemy = enemies[Random.Range(0, enemies.Length)];
            float location = Random.Range(-5f,5f);
            Instantiate(enemy,new Vector3(location,5f,0f),Quaternion.identity);

            //increasing difficulty after every spawn
            timeBetweenSpawns -=decreaseAmt;
            if (timeBetweenSpawns < minSpawnTime) {
                timeBetweenSpawns = minSpawnTime;
            }

            spawnTimer = timeBetweenSpawns;
        } else {
            spawnTimer -= Time.deltaTime;
        }

    }
}
```

If the **`spawnTimer`** is 0, spawn a new enemy, otherwise, decrease the **`spawnTimer`**.

Volcano Island: Spawner Script

```
    // Update is called once per frame
    void Update()
    {
        if (spawnTimer <= 0) {
            //pick random enemy prefab
            GameObject enemy = enemies[Random.Range(0, enemies.Length)];
            float location = Random.Range(-5f,5f);
            Instantiate(enemy,new Vector3(location,5f,0f),Quaternion.identity);

            //increasing difficulty after every spawn
            timeBetweenSpawns -=decreaseAmt;
            if (timeBetweenSpawns < minSpawnTime) {
                timeBetweenSpawns = minSpawnTime;
            }

            spawnTimer = timeBetweenSpawns;
        } else {
            spawnTimer -= Time.deltaTime;
        }

    }
}
```

Pick an enemy prefab from the **`enemy GameObject`** and pick a random **`location`** for the x position when spawned

Volcano Island: Spawner Script

```
    // Update is called once per frame
    void Update()
    {
        if (spawnTimer <= 0) {
            //pick random enemy prefab
            GameObject enemy = enemies[Random.Range(0, enemies.Length)];
            float location = Random.Range(-5f,5f);
            Instantiate(enemy,new Vector3(location,5f,0f),Quaternion.identity);

            //increasing difficulty after every spawn
            timeBetweenSpawns -=decreaseAmt;
            if (timeBetweenSpawns < minSpawnTime) {
                timeBetweenSpawns = minSpawnTime;
            }

            spawnTimer = timeBetweenSpawns;
        } else {
            spawnTimer -= Time.deltaTime;
        }

    }
}
```

The **`Instantiate()`** method is how you can take a prefab and put it into the world via code. There are many overloads for this method.

Volcano Island: Spawner Script

```
    // Update is called once per frame
    void Update()
    {
        if (spawnTimer <= 0) {
            //pick random enemy prefab
            GameObject enemy = enemies[Random.Range(0, enemies.Length)];
            float location = Random.Range(-5f,5f);
            Instantiate(enemy,new Vector3(location,5f,0f),Quaternion.identity);

            //increasing difficulty after every spawn
            timeBetweenSpawns -=decreaseAmt;
            if (timeBetweenSpawns < minSpawnTime) {
                timeBetweenSpawns = minSpawnTime;
            }

            spawnTimer = timeBetweenSpawns;
        } else {
            spawnTimer -= Time.deltaTime;
        }

    }
}
```

The version we use takes a **`GameObject`**, a **`Vector3`** position, and a rotation (a **`Quaternion`** equivalent to 1 in this case).

Volcano Island: Spawner Script

```
    // Update is called once per frame
    void Update()
    {
        if (spawnTimer <= 0) {
            //pick random enemy prefab
            GameObject enemy = enemies[Random.Range(0, enemies.Length)];
            float location = Random.Range(-5f,5f);
            Instantiate(enemy,new Vector3(location,5f,0f),Quaternion.identity);

            //increasing difficulty after every spawn
            timeBetweenSpawns -=decreaseAmt;
            if (timeBetweenSpawns < minSpawnTime) {
                timeBetweenSpawns = minSpawnTime;
            }

            spawnTimer = timeBetweenSpawns;
        } else {
            spawnTimer -= Time.deltaTime;
        }

    }
}
```

After each spawn, decrease the time between spawns (so the next fireball will spawn sooner) to a minimum value and reset the **`spawnTimer`**.

What we still need (for today anyway)

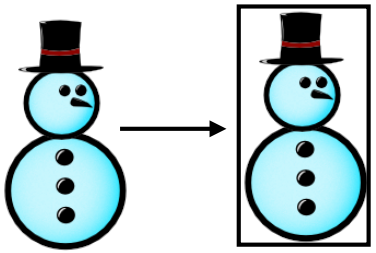
- Collision for the snowman and enemies (to collide with the ground, walls, and each other)
- Damage system that interacts with the snowman health
- A simple way to reset the game.

Particle effects, animations, sounds, and an intro/ game over screen are on the list too, but we'll work on them after we have the basic mechanics down.

Working on "polish" should take a back seat to a working (and enjoyable) game.

Colliders

Colliders are components that define the shape (and parameters) used to determine if two objects have "touched".



Exactly what happens when objects "touch" depends on what you code and other attached components (like **`Rigidbody`**).

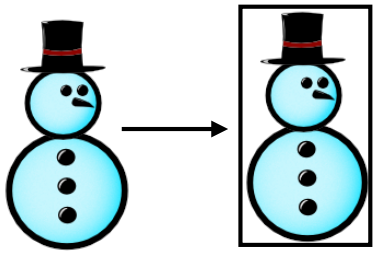
Most games make the collider smaller than the actually graphic...

Typically, you'll want to use simple shapes, it makes the behind-the-scene calculations faster.

Colliders

- Add a BoxCollider2D component to your player object.
- Adjust the collider size by clicking on the "Edit Collider" button 

- Make sure the collider is appropriately positioned and sized for your player.
- Do the same for the Enemies using Box or Circle colliders



Colliders

- Create a new empty gameobject and name it "RightWall".
- Move it to the right side of the play area and add a BoxCollider2D.

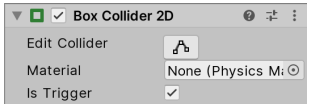
This will represent the right-side limit to the play area. Adjust the collider so that the player will not be able to move beyond the limits of the screen.

This will work fine for a game with a fixed aspect ratio. For different ratios, you will need to create other scripts to adjust the "wall" position

Do the same for a left wall and the ground, where you want the fireballs to appear to strike the ground.

Colliders (one last thing)

- Open the enemy prefab, which should already have a BoxCollider2D.
- Check the "is Trigger" checkbox.



This creates a trigger rather than a physical collider. We'll see why this is useful shortly.

It's perfectly fine to use a regular physical collider here, but our code will be slightly different.

This is the only trigger we'll have in this project. In future games, we'll use them more extensively.

Player Reset

Player Class

```
public void reset(){
  health=3;
  Vector3 pos =new Vector3(0f,-1.41f,0f);
  this.transform.position=pos;
  this.gameObject.SetActive(true);
}
```

This sets a view variables back to their initial starting values, so that the game can be played again.

Resets the players health back to the initial value. The actual value should not be hard coded.

Player Reset

Player Class

```
public void reset(){
  health=3;
  Vector3 pos =new Vector3(0f,-1.41f,0f);
  this.transform.position=pos;
  this.gameObject.SetActive(true);
}
```

This sets a view variables back to their initial starting values, so that the game can be played again.

This is a position that make sense for the initial location of the snowman, this can be whatever you want.

Player Reset

Player Class

```
public void reset(){
  health=3;
  Vector3 pos =new Vector3(0f,-1.41f,0f);
  this.transform.position=pos;
  this.gameObject.SetActive(true);
}
```

This sets a view variables back to their initial starting values, so that the game can be played again.

`SetActive()` is equivalent to selecting the checkmark next to the object name in the inspector. When set to **`false`**, the gameobject, and any children, do not appear in the scene. This can be an easy way to remove an object without destroying (deallocating) it.

Spawner Reset

Spawner Class

```
public void reset(){
  timeBetweenSpawns=1.25f;
}
```

When the spawner gets reset, the “difficulty” of the game should also reset. Again, this value should not be hard coded, but grabbed from a public variable.

After the game ends, the player will have an choice, play again or return to the menu. Playing again will call both the Player **`reset()`** method and the Spawner **`reset()`** method.

Starting GameManager

```
public class GameManager : MonoBehaviour
{
    private static GameManager _instance = null;

    void Awake(){
        if(_instance == null){
            _instance = this;
        }
    }

    public static GameManager instance(){
        return _instance;
    }
}
```

Singleton portion, see the previous lecture.

Starting GameManager

```
public class GameManager : MonoBehaviour
{
    private static GameManager _instance = null;

    void Awake(){
        if(_instance == null){
            _instance = this;
        }
    }

    public static GameManager
        return _instance;
    }
}
```

The GameManager will contain code for dealing with the UI, mouse clicks on menus, resetting the game, etc. Often times, a Manager type class will help facilitate communication between classes.

Singleton portion, see the previous lecture.

Starting GameManager

There are two ways to get references to the player and spawner within the GameManager.

Create public variables for **`Player`** and **`Spawner`** and use the inspector.

Use a **`Find`** method

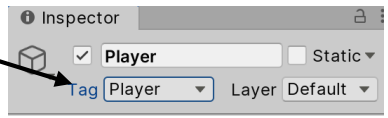
```
void Start()
{
  player = GameObject.FindGameObjectWithTag("Player").
                                        GetComponent<Player>();
  spawner = GameObject.FindGameObjectWithTag("Spawner").
                                      GetComponent<Spawner>();
  player.reset();
  spawner.reset();
}
```

`Find` Version

This uses the **`Tag`** version of find, so we *must* use tags...

Starting GameManager

Setting a tag for a gameobject is easy, but there are only a few build in tags for you to use.



You can add more tags, however, by opening the Tags & Layers panel in the inspector ("Add Tags" in the dropdown)

Add and set as many as you want, then the **`Tag`** version of **`Find`** will work correctly.

Typically, finds are slow, don't do it in methods that are called every frame!

Damage

Player Class

```
public void takeDamage(int value) {
  health -= value;

  if (health <= 0) {
  //player dies
  this.gameObject.SetActive(false);
  }
}
```

Within the player class, this is an easy way to subtract from the player health.

Any time the health is less than or equal to 0, we'll use **`SetActive()`** to make the object disappear.

Later on, this method will also spawn particle effects and inform the GameManager that UI should be updated...

Damage

```
private void OnTriggerEnter2D(Collider2D collision) {
  if (collision.tag == "Player") {
    //reduce player health
    p.takeDamage(damage);

    GameObject.Destroy(gameObject);
  }
}
```

Enemy Class

Remember, the Enemy has a collider component set to a trigger.

So, when that collider touches another collider **`OnTriggerEnter2D()`** will be called.

The collision parameter is a reference to the other collider that was touched. If it's **tag** is "Player", we'll have the player take damage and destroy the enemy (it'll despawn).

Damage

```
private void OnTriggerEnter2D(Collider2D collision) {
  if (collision.tag == "Player") {
    //reduce player health
    p.takeDamage(damage);

    GameObject.Destroy(gameObj
  }
}
```

Notice that we do require a reference to the player object to do this!

Enemy Class

Remember, the Enemy has a collider component set to a trigger.

So, when that collider touches another collider **`OnTriggerEnter2D()`** will be called.

The collision parameter is a reference to the other collider that was touched. If it's **tag** is "Player", we'll have the player take damage and destroy the enemy (it'll despawn).

Damage

Design Note:

Should the enemy have a reference to the player?

For a small game, it may be ok, but this sort of dependency *can* cause problems down the road. Whenever you add a reference to another class/object to your script, ensure that it makes sense.

Another way to do this example is to send all communication through the GameManger (the enemy tells the manager that the player took damage).

be called.

```
llider2D collision) {
 {
```

Notice that we do require a ference to the player object to do this!

Enemy Class

The collision parameter is a eference to the other collider hat was touched. If it's **tag** is Player", we'll have the player ake damage and destroy the enemy (it'll despawn).

Coming up

- Unity Particle System
  - Standard Particle System
  - VFX Graph