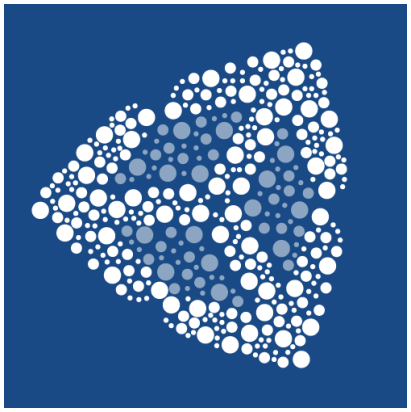
ImpactLab - Game Dev
Lecture 4: More VI and VFX in Unity

Summer 2024

School of Computing
and Data Science

Wentworth Institute of
Technology



What we still need (for today anyway)

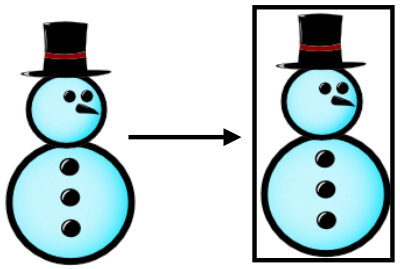
- Collision for the snowman and enemies (to collide with the ground, walls, and each other)
- Damage system that interacts with the snowman health
- A simple way to reset the game.

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`**).

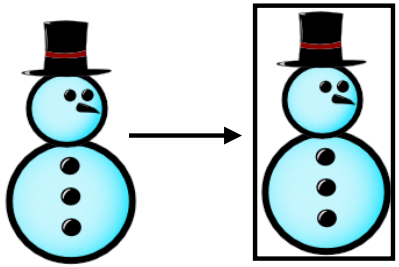
Most games make the collider smaller than the actually graphic...

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.

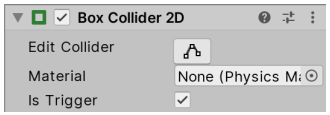
This will represent the right-side limit to the play area. Adjust the collider so that the player will not be able to move beyond the limits of the screen.

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.

a Science

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.

Wentworth
Computing & Data Science

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.

Wentworth
Computing & Data Science

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.

Wentworth
Computing & Data Science

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();
```

Wentworth
Computing & Data Science

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();
}
```

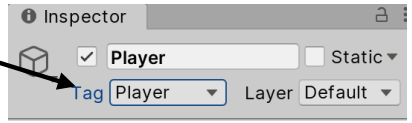
Find Version

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

Wentworth
Computing & Data Science

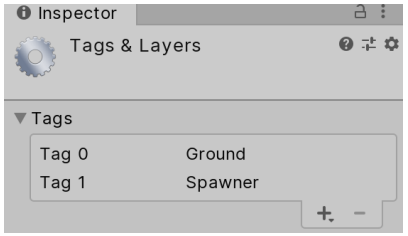
Starting GameManager

Setting a tag for a gameobject is easy, but there are only a few built 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...

Wentworth
Computing & Data Science

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).

```
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).

Particle Systems

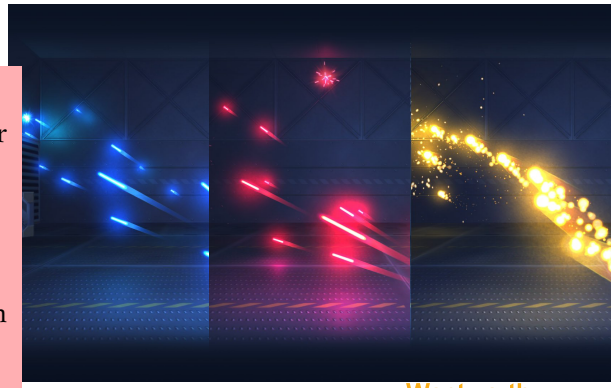
- In Unity, Particle Systems are components that you add to game objects.
- Typically, you'll create a single gameobject for each particle system and either instantiate the object (to play the system) or attach it as a child to another object.

Learning about the Unity particle system component is mostly about playing around with settings and/or trying to reproduce effects you've seen in other games.

There are way too many options and settings to explore all of them in a single lecture.

Particle Systems

- Particle systems are typically used as visual effects representing everything from fire and smoke to sparks to weapon fire.



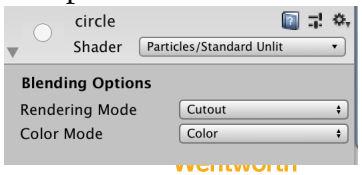
There are many tutorials out there for creating specific particle systems.

Today, we'll create a basic system and then reproduce a partial effect from a game.

Particle Systems

- In an empty Unity project, create a new empty gameobject and add a new particle system.
- The actual particles need a sprite or a mesh (essentially a material) to display properly.
  - Create a new material, change the shader to "Particles/Standard Unlit", Render mode to Cutout, Color mode to Color and select one of the UI sprites for the Albedo.

We're using a simple, builtin sprite for this example. In the future, we'll create our own.



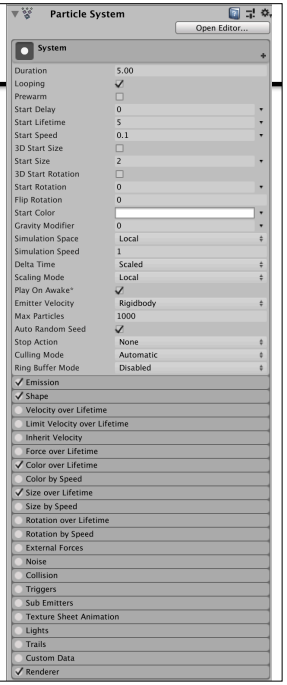
Particle Systems

In each particle system there is an initial area for settings. Below that are other modules that can be activated to provide different options for how the particles behave.

You'll usually always use "Emission", "Shape" and "Renderer". The others will depend on the situation.

The Renderer module is where we will apply are material to the particles.

There are way too many settings and modules to talk about all of them. Experiment with them to come up with something cool.

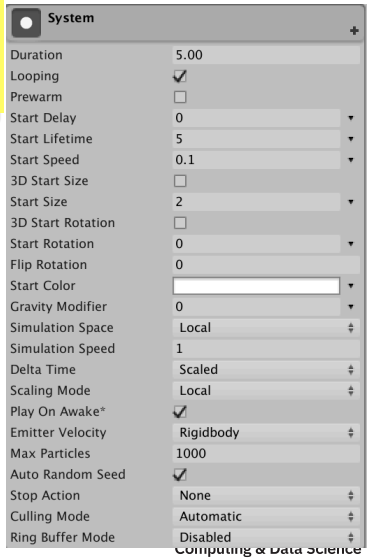


Particle Systems

The main module gives many options related to how the particle system will behave.

In the screenshot, I've set a few of the values, but I encourage you to play with them to see what happens.

The drop down arrays next to the settings allow you to use random values and graphs to create more variability in the particle look and behavior.



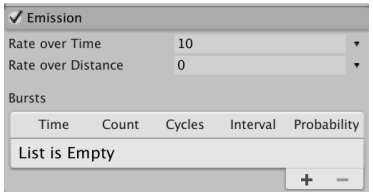
Particle Systems

For the first example we'll use a few of the modules:

Emission:

Controls how the particles are spawned.

We'll use a constant rate over time, but you can also do a burst spawn.



Color over Lifetime:

Changes the tint and alpha of the particles as time goes by. This is typically done using a gradient.

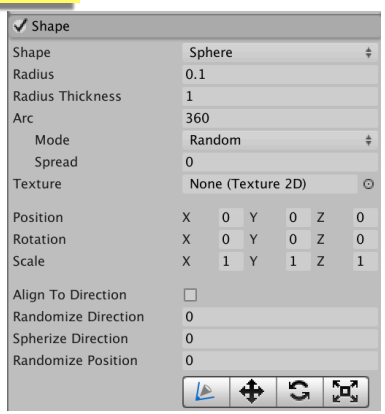
Particle Systems

For the first example we'll use a few of the modules:

Shape:

Controls where the particles are spawned.

Different shapes lead to different movement and spawn patterns. For en explosion, a sphere or hemisphere is usually best.



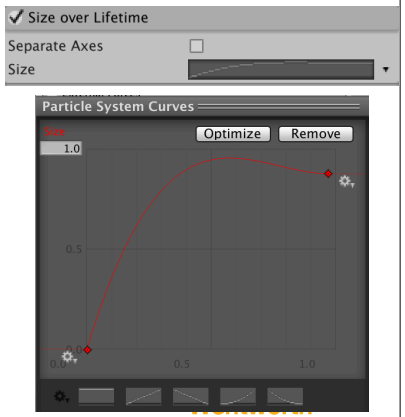
Particle Systems

For the first example we'll use a few of the modules:

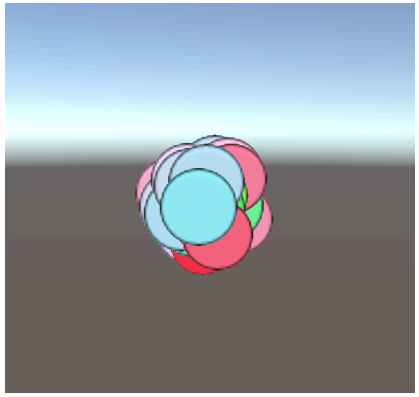
Size over Lifetime:

Controls the size of the particles as time goes by. Usually controlled via a graph.

This is something that you can play with to get the desired effect. Most particles will "fade out" using a combination of shrinking and increasing alpha.



Particle Systems



Even though this is a simple system, it uses many of the features that are necessary for more complex particle effects.

Most effects that you see in games are combinations of multiple particle systems.

Visual Effects Graph

The Visual Effects Graph is a drag-and-drop, node based tool, built into Unity, that allows us to create complex particle effects very easily.

It's a complex tool! Just like with the regular particle system, you should experiment with different settings to see what you can create.

Notes:

Only fully available on Unity 2019.3 or greater

Uses the GPU to simulate particles, so it's not always applicable for your target platform.

It can also be a laptop battery hog!

The feature must be added to your project:

Window -> Package Manager

Search for Visual Effects Graph and install.

Visual Effects Graph

The Visual Effects graph contains several "Context" areas:

Spawn: Spawns the Particles — How Many Particles? When are they Spawned?

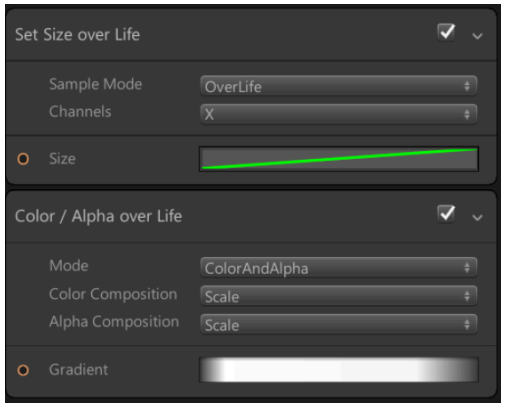
Initialize: Starting Values — Longevity? Where are they Spawned? Velocity?

Update: Behavior over Time — Forces? Color Change?

Quad Output: Rendering The Particles — What do they look like? Shader? Facing?

Visual Effects Graph

Within each Context, there are a number of blocks, of which you can add more or delete.



Each block sets a specific property related to the context. Many are the same as what you see in the regular Unity particle system.

Visual Effects Graph

The Visual Effects Graph uses the GPU to render particles whereas the standard particle system uses the CPU. This has several important consequences:

CPU vs GPU Particles

	CPU Particle System	GPU Visual Effect Graph
Particle Count	Thousands	Millions
Simulation	Simple	Complex
Physics	Underlying Physics System	Primitives, Depth Buffer, Scene Representation (E.g. SDF)
Gameplay Readback	Yes	No* *Potentially small data with latency
Other	-	Can read frame buffers

Visual Effects Graph

Now we're going to play around and create a few effects. Keep in mind, this can severely impact your laptop battery!

Additional Resources:

- For more examples and VFX Graph information, take a look at the Unity Blog. e.g. https://blogs.unity3d.com/2019/03/06/visual-effect-graph-samples/
- The Brackys YouTube channel also has several good Visual Effects Graph tutorials (smoke and fireworks)

Volcano Island: What's Left?

- Particle Effects
- Animation
- Sound/Music
- Start/Game Over Screen

I've provided some assets for you to help with creating this items. As we go through this list, feel free to experiment with the settings.

Particle Effects

We'll start by making a new gameobject and adding a particle system component.

Drag the gameobject into your Prefabs folder. In the future, we'll instantiate this object when we need to spawn the particle effect.

The goal is to use three effects:

When the fireball hits the ground.

When the fireball hits the player.

When the player dies.

We'll keep the effect simple for now. Later, you may want to increase the fidelity.

Instantiating the Particle Effects

Make sure you have a couple of effects ready to go, specifically the ones for the enemy hitting the ground and the player.

Enemy Class

```
//death particle effects
public GameObject deathEffect;

private void OnTriggerEnter2D(Collider2D collision) {
  if (collision.tag == "Player") {
    //reduce player health
    p.takeDamage(damage);
    GameObject.Destroy(gameObject);
  }
  if (collision.tag == "Ground") {
    Instantiate(deathEffect, new
Vector3(transform.position.x, transform.position.y-0.5f,
-0.3f), Quaternion.identity);
    GameObject.Destroy(gameObject);
  }
}
```

ce

Instantiating the Particle Effects

Make sure you have a couple of effects ready to go, specifically the ones for the enemy hitting the ground and the player.

Enemy Class

```
//death particle effects
public GameObject deathEffect;

private void OnTriggerEnter2D(Collider2D collision) {
  if (collision.tag == "Player") {
    //reduce player health
    p.takeDamage(damage);
    GameObject.Destroy(gameObject);
  }
  if (collision.tag == "Ground") {
    Instantiate(deathEffect, new
Vector3(transform.position.x, transf
-0.3f), Quaternion.identity);
    GameObject.Destroy(gameObject);
  }
}
```

Ensure that you drag your death effect prefab into this script in the inspector.

Don't forget to add a new tag, "Ground" and set the ground object to it.

Instantiating the Particle Effects

Enemy Class

```
//death particle effects
public GameObject deathEffect;
public GameObject hitEffect;

private void OnTriggerEnter2D(Collider2D collision) {
  if (collision.tag == "Player") {
    //reduce player health
    p.takeDamage(damage);
    Instantiate(hitEffect, new
Vector3(transform.position.x, transform.position.y-0.3f,
-0.3f), Quaternion.identity);
    GameObject.Destroy(gameObject);
  }
  if (collision.tag == "Ground") {
    Instantiate(deathEffect, new
Vector3(transform.position.x, transform.position.y-0.5f,
-0.3f), Quaternion.identity);
    GameObject.Destroy(gameObject);
  }
}
```

Instantiating the Particle Effects

Enemy Class

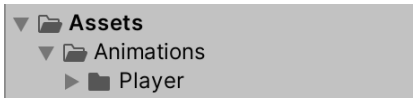
```
//death particle effects
public GameObject deathEffect;
public GameObject hitEffect;

private void OnTriggerEnter2D(Collider2D collision) {
  if (collision.tag == "Player") {
    //reduce player health
    p.takeDamage(damage);
    Instantiate(hitEffect, new
Vector3(transform.position.x, transform.position.y-0.3f,
-0.3f), Quaternion.identity);
    GameObject.Destroy(gameObject)
  }
  if (collision.tag == "Ground") {
    Instantiate(deathEffect, new
Vector3(transform.position.x, tran
-0.3f), Quaternion.identity);
    GameObject.Destroy(gameObject);
  }
}
```

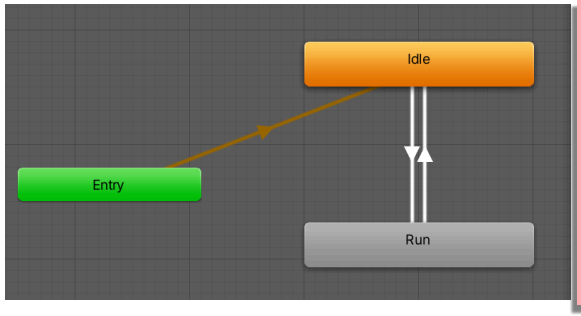
Do the same thing with the hitEffect (the particles that play when the player is hit by the fireball).

Animation

- Let's keep organized:



Animation in Unity uses a *state machine* to control transitions to different animation states



Here, the starting animation is Idle (transition upon Entry).

Under certain conditions, the animation will transition to Run and/or back to Idle.

Animation

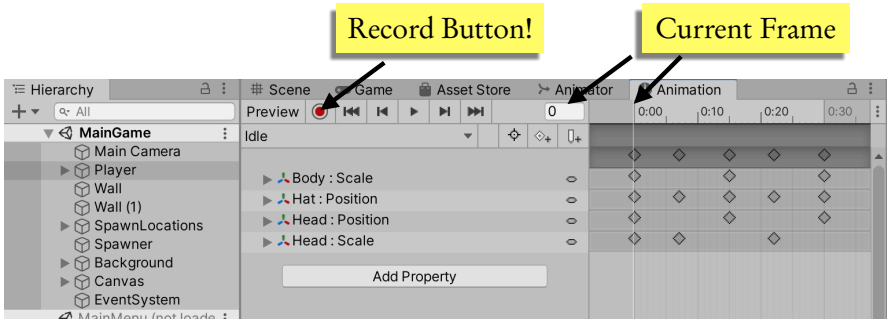
- Create a new animation controller (Right Click in the Project window -> Create -> Animation Controller), Rename to PlayerAnimation.
- Create two animations (Right Click in the Project window -> Create -> Animation), Rename to Idle and Run.

The animation controller gives you the state machine for controlling transitions.

The two animations are where we will record changes to properties of our snowman that will play when the controller is in a specific state.

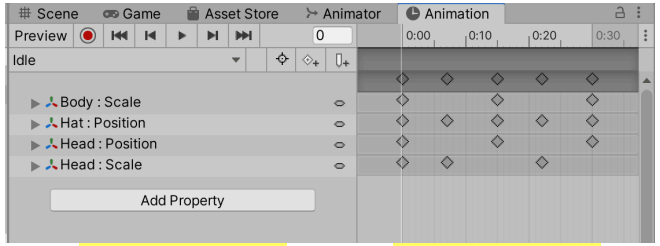
Animation: Idle

- We're going to start by opening the Animation window, attaching it to one of the frames and selecting our top-level snowman object.



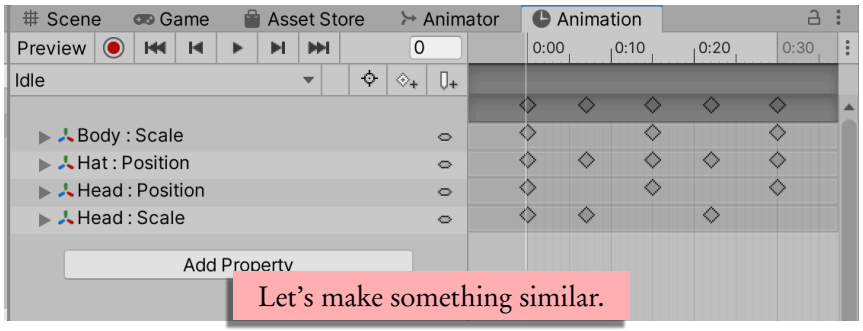
Animation: Idle

- To animate: Set a property to an initial value (the first keyframe) then add more keyframes and change the value (like the scale of the Head object).
- Unity will interpolate the position of each frame in between and play it like an animation



Let's make a head banging snowman

Animation: Idle

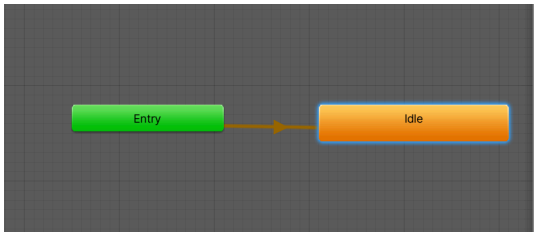


Let's make something similar.

This is my idle animation, notice that I change the scale and position of different pieces of the snowman.

To make a repeating animation, I copy the initial keyframes at 0 and move them down the timeline, then I make changes in between.

Animation: State Machine



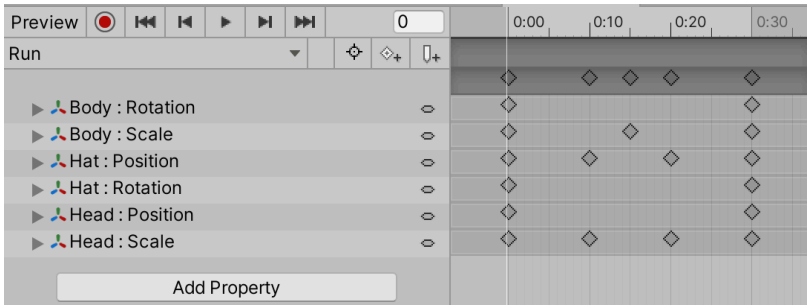
State Machines are used to keeping track of objects that can be in one of any number of "state".

Unity uses this idea of animations. At any given time, the object will be playing a specific animation.

The basic state machine will have an Entry state (for when the object is created), this will connect to one of the animations that you have created (Idle here)

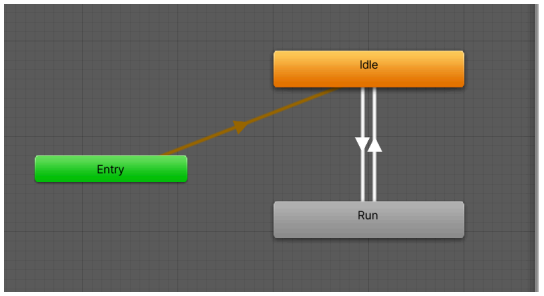
Animation: Run

For the Run animation, I'm going to have the snowman lean forward (in the direction we are running). In addition, it will still do some of the Idle style scaling and transforming.



I change more properties here, so let's try to make something similar.

Animation: State Machine Again



To add more animation states, just drag the animation unto the state machine interface.

We'll add the transitions in just a moment (the white arrows). In general, you'll have transitions between states that are triggered by variables or functions in your code.

Animation: Transitions

- Transitions are how we tell Unity to change from one animation to another.
- Typically this is done through some kind of trigger.
- The triggers are associated with an Animator component that has the Animation Controller as a property.

Add an Animator to the Player gameobject.

Drag the AnimationController that you created to the appropriate place in the Animator.

Now we are ready to go, any animation that we add to our controller will be hooked up to the player object.

Animation: Transitions

- Setting up a transition is as easy as selecting a block, right clicking and selecting Make Transition.
- This attaches an arrow to your mouse cursor which allows you to select another block.
- The transition will go from where you first right clicked to where you clicked.
- Once the transition is created, highlight it.

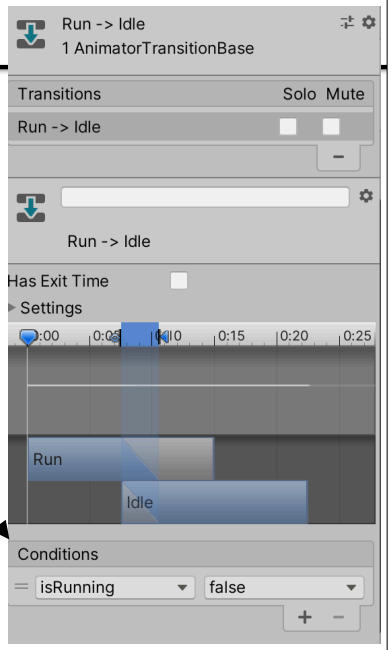
Animation: Transitions

This is my transition from Run to Idle.

There is some info about Exit Time and a timeline graph.

For now the important part is the *Conditions*.

This transition is triggered when the isRunning (boolean) is false, which we will set in our code.

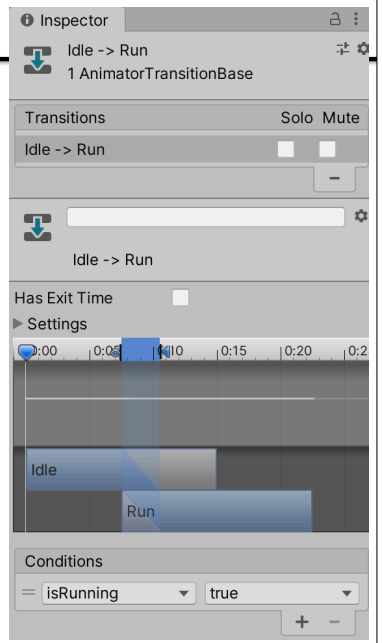


Animation: Transitions

Similarly, the transition from Idle to Run use the same boolean, isRunning.

The timeline shows the amount of time that the transitions between states will take. Play with it to see what happens.

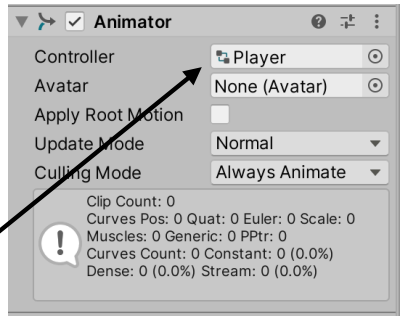
This transition is triggered when the isRunning (boolean) is true, which we will set in our code.



Animation: Attaching to Object

Lastly, we need to attached the animation controller to the object that will be animating.

On your snowman, add the Animator component and drag the animation controller (Player in this case) into the Controller location.



When you play the game, the snowman should play the Idle animation. Next we need to hook up the running animation into the code.

Hooking up the Code

Player Class

```
void Start()
{
  ...
  anim = GetComponent<Animator>();
}

private void Update() {
  if (input > 0 || input<0) {
    anim.SetBool("isRunning", true);
  } else {
    anim.SetBool("isRunning", false);
  }

  if (input > 0) {
    transform.eulerAngles = new Vector3(0f, 0f, 0f);
  }
  else if(input < 0){
    transform.eulerAngles = new Vector3(0f, 180f, 0f);
  }
}
```

We must grab the animator component

Depending on the input (from FixedUpdate) we set the isRunning bool to true or false.

What does this do?

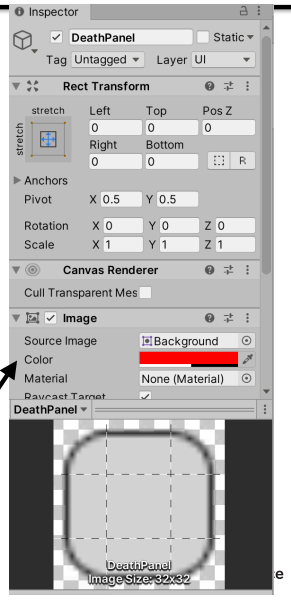
Game Over Screen

We're going to be using the standard Unity UI system to create our Main Menu and Game Over screen

Start by creating a new UI->Image in your Hierarchy.

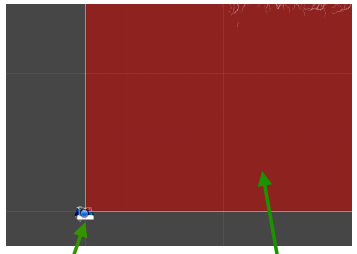
This automatically creates a Canvas for you that contains the Image object as a child.

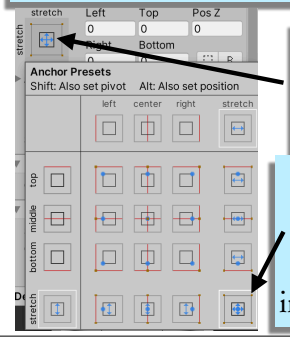
We'll have to set the image that will be displayed. Initially, we'll set the tint to a solid color (with transparency). I'm using the standard button image that is build in.



Game Over Screen

In your scene view, the canvas will look huge. Don't worry, it will display properly when you run the game. However, we'll have to mess with the anchor settings to get it to look how we want.



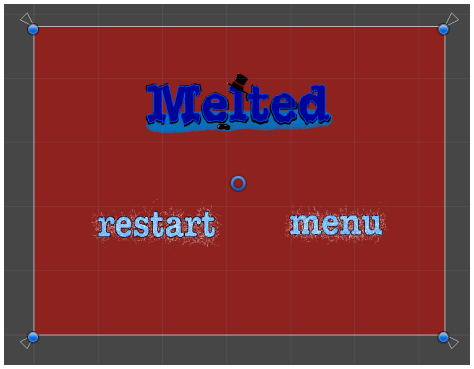


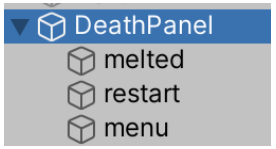
Click here to see the Anchor Presets

Notice that I have the stretch selected (in both dimensions. Using modifier keys (shift and alt) will set different anchor properties. Add more images or text and play around with the settings.

Game Over Screen

Here's an example:



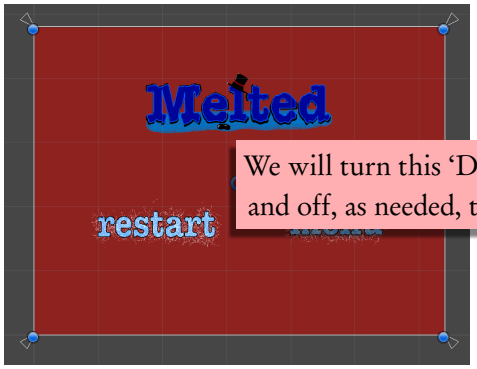


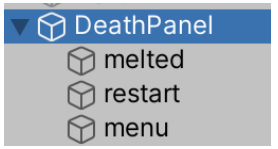
All of the objects are children to the panel that contains the background color (which itself is a child of the canvas).

'Melted' is just an image, but 'Restart' and 'Menu' contain an image and a button component. It's easy to add more components to your UI elements.

Game Over Screen

Here's an example:





We will turn this 'DeathPanel' on and off, as needed, through code.

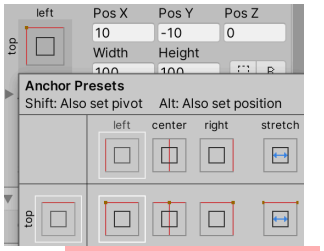
e objects are
the panel that
contains the background color (which itself is a child of the canvas).

'Melted' is just an image, but 'Restart' and 'Menu' contain an image and a button component. It's easy to add more components to your UI elements.

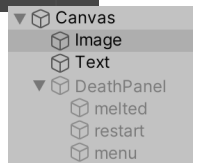
Health UI

Here's the Health UI on the same Canvas:

Note that the anchor is in the top left:



And the Image and Text are children of the Canvas (not the DeathPanel):



We'll update the text via code, for now I've entered "x3" as the default.

Code Hookup

Remember: public fields are updated in the inspector.

```
public Text healthText;
public GameObject DeathPanel;

public void updateHealthText(int health){
  healthText.text = "x" + health
}

public void deathPanelSwitch(bool state){
  deathPanel.SetActive(state)
}
```

GameManager.cs

These are the public methods that allow us to update the GUI information, i.e. show the DeathPanel and change the text for the health.

Code Hookup

Now we can add the previous calls to the appropriate places in our current code:

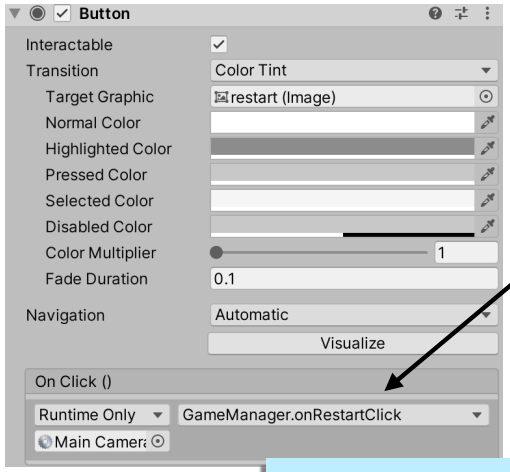
GameManager.cs

```
void Start(){
  deathPanelSwitch(false);
  ...
}

public void onRestartClick(){
  if(deathPanel.activeSelf){
    deathPanelSwitch(false);
    player.reset();
    spawner.reset()
  }
}
```

And create the method to call when the restart button is pressed

Code Hookup



Easy to hook up function calls to button presses.

Restart Button

Back to Player and Spawner

Our onRestartClick() method resets the player and the spawner:

Position the player wherever you want, and ensure that the player gameObject is active.

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

Player.cs

Spawner.cs

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

Reset spawner timing.

Back to Player and Spawner

When the player takes damage and/or “dies” we need to make sure we trigger the GameOver screen

```
public void takeDamage(int value){
  health-=value;
  GameManager.instance().updateHealthText(health);
  if(health<=0){
    //Instantiate death effect here if you want
    this.gameObject.setActive(false);
    GameManager.instance().deathPanelSwitch(true);
  }
}
```

This will make the player gameobject disappear (without destroying it in memory) and show the “DeathPanel”

Coming Up:

- Finishing Volcano Island