ImpactLab - Game Dev
Lecture 6: Finishing Up VI

Summer 2024

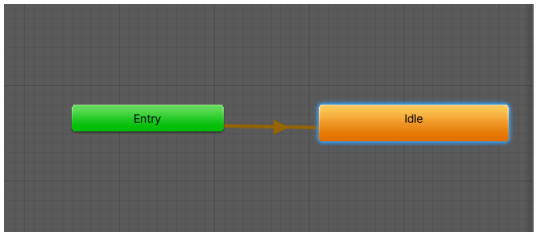
School of Computing and Data Science

Wentworth Institute of Technology

Volcano Island

Wentworth
Computing & Data Science

Animation: State Machine



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

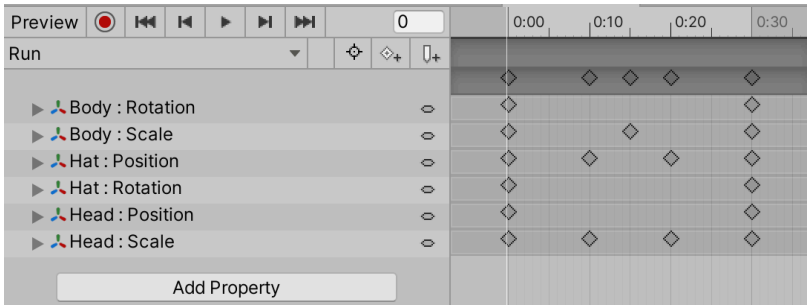
Unity uses this idea of animations. At any given time, the object will be playing a specific animation.

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)

Wentworth
Computing & Data Science

Animation: Run

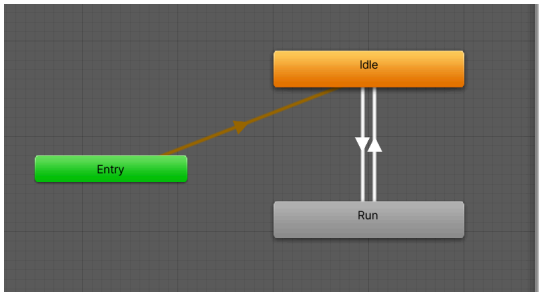
For the Run animation, I'm going to have the snowman lean forward (in the direction we are running). In addition, it will still do some of the Idle style scaling and transforming.



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

tworth
ting & Data Science

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.

Wentworth
Computing & Data Science

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.

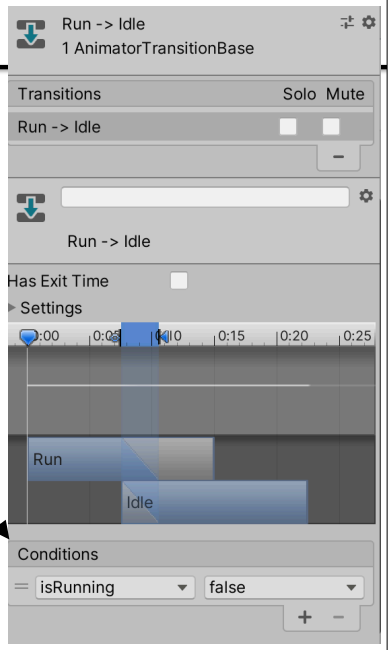
Animation: Transitions

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.

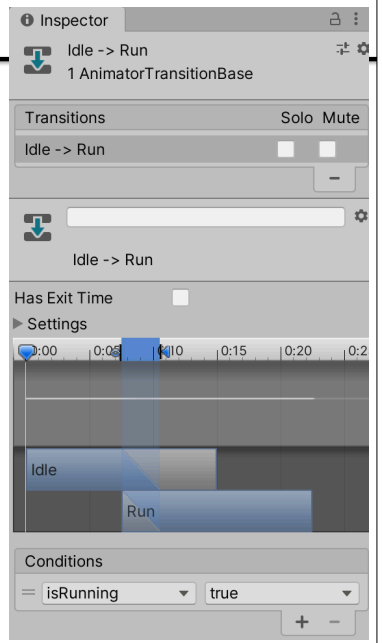


Animation: Transitions

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.

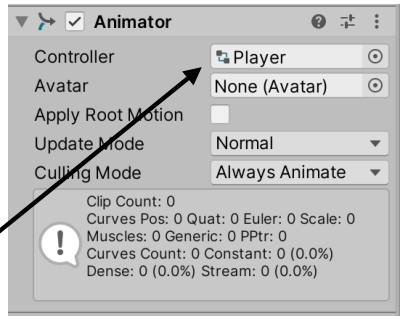
This transition is triggered when the isRunning (boolean) is true, which we will set in our code.



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?

What's Left?

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

Our last two items are sound effects/Music and a Start and Game Over Screen.

These will require some classes that manage the state of our game.

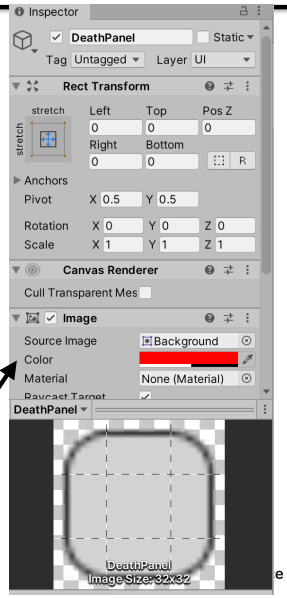
Game Over Screen

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.

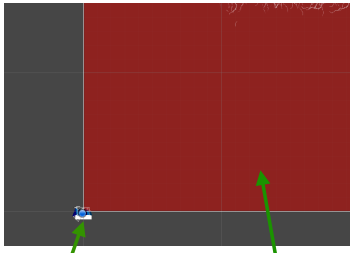
This automatically creates a Canvas for you that contains the Image object as a child.

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.

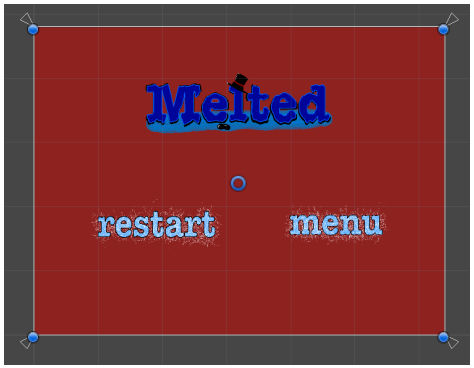


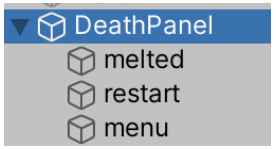
Click here to see the Anchor Presets

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:



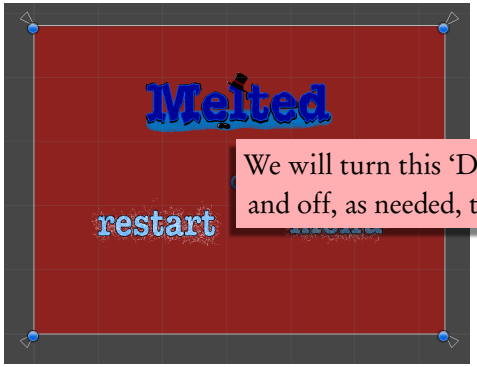


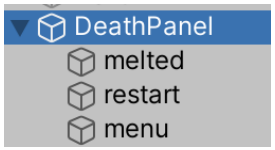
All of the objects are children to the panel that contains the background color (which itself is a child of the canvas).

'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.

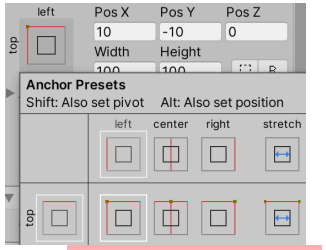
e objects are
the panel that
contains the background color (which itself is a child of the canvas).

'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.

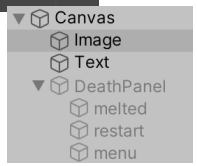
Health UI

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

```
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

Remember: public fields are updated in the inspector.

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:

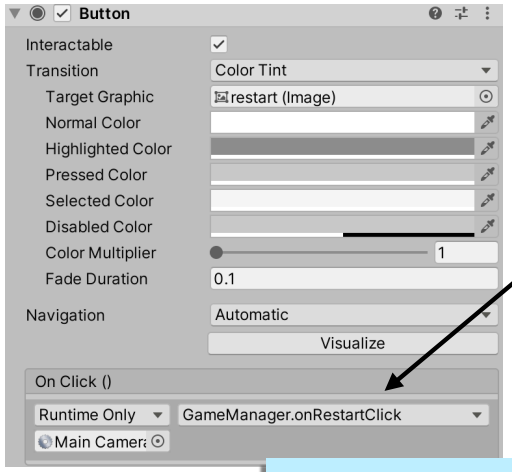
GameManager.cs

```
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"

Lots of Little Stuff

OK! There's always lots of little questions that we have to answer, for example:

Q: Do we want fireballs to continue spawning after the players dies?

A: Add a guard clause in the spawner script update() to check if the player is active.

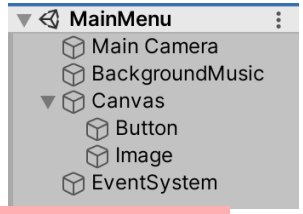
Q: How do we ensure the health is correct and the DeathPanel is off at the start of the game?

A: In the GameManager start(), call reset() on the player and spawner and call deathPanelSwitch(false)

Main Menu

Create a new Scene, call it MainMenu and put it into the scenes folder. Load this scene (double click it).

Ensure that there is a Camera in the scene and add a UI button and UI Image.



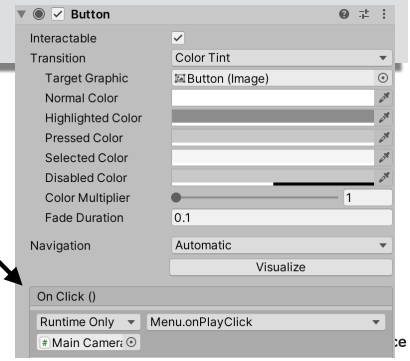
We'll add that BackgroundMusic gameobject in a moment

Main Menu

Create a new script called Menu.cs and attach it to the Main Camera

```
public class Menu : MonoBehaviour
{
  public void onPlayClick(){
    SceneManager.LoadScene("MainGame");
  }
}
```

Hook this up to the button that you just created. This loads the scene "MainGame" and unloads the current scene.



Back in GameManager and the MainGame Scene

Create a method to 'fire' if the user presses the menu button from the GameOver screen:

```
public void onMenuClick(){
  if(deathPanel.activeSelf){
    SceneManager.LoadScene("MainMenu");
  }
}
```

Hook this up to the menu button on the DeathPanel.

This is how you change scenes.

Scenes can represent many things: Menus, levels, cutscenes, etc.

Music and Sound

For now, we're just going to add the background music. Adding sound effects will be up to you in the homework.

We'll add the new gameObjects to the MainMenu scene.

Then, we'll set it to persist across scene transitions.

Sound effects will work similarly to what you see here, though they will likely just play once.

Feel free to use whatever software you want for creating music and sound effects.

Music:

- Bosca Ceoil
- LMMS

SoundFX:

- Audacity+ your voice
- Bfxr

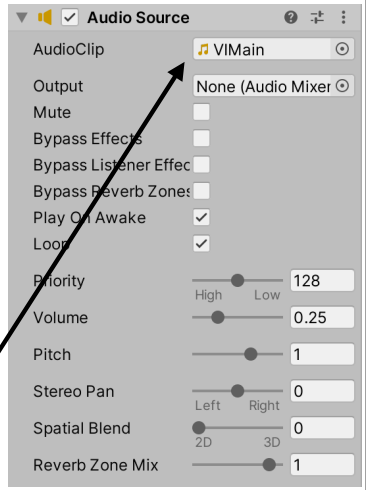
Background Music

Let's start by creating a new empty gameobject, name it "Background Music" and add the "Audio Source" component.

Make sure to have Play on Awake and Loop enabled.

Now create a new script, MusicManager and add it to the gameobject that we just created.

Since this will be our background music, drag your audio clip here



Background Music

We'll start by making the MusicManager class a singleton and set it so that changing scenes doesn't destroy the object (so the music keeps playing).

```
[RequireComponent(typeof(AudioSource))]
public class MusicManager : MonoBehaviour
{
  AudioClip backgroundClip;

  static MusicManager _instance=null;

  void Awake(){
    if(_instance==null){
      _instance=this;
    } else {
      Destroy(this.gameObject);
    }
    DontDestroyOnLoad(this.gameObject);
  }
```

MusicManager Class

Background Music

The start method is what grabs a reference to the AudioSource component and sets the backgroundClip. Then, if the clip is not playing, play it.

```
void Start()
{
  AudioSource source=GetComponent<AudioSource>();
  backgroundClip=source.clip;

  if(!source.isPlaying){
    source.Play();
}
```

MusicManager Class

Coming Up:

- Shader Graph