Unity notes

**Roll-a-Ball**

* Press F to focus perspective on the selected object
* Add Rigidbody for in game physics
* Ctrl+B for build settings
* Player input settings
  + Create Input Action Asset and link certain keys to an action name
  + Add the new script component to an object
* Writing scripts
  + A function is a way to group code under one name
  + Two functions included in the template, Start and Update
  + Update is called before rendering a frame and this is where most of your code will go
  + Fixed update is called just before performing any physics calculations and this is where your physics code will go
  + Name spaces are a collection of classes and other datatypes which can be imported at the start of your script
  + Already three name spaces in the template: System.Collections, System.Collections.Generic, UnityEngine
  + Add a new one called UnityEngine.InputSystem; for inputs
  + Void means this function will do a task and won’t return any values
  + Write void OnMove()
  + Inside the parentheses, add InputValue (type of variable) and movementValue (name of the variable you will use in the function)
  + The movement value variable will capture and store the data from the x and y direction of input, this kind of data can be stored as a vector2 variable
  + Write Vector2 movementVector = movementValue.Get<Vector2>() in the parentheses; this code takes or gets the vector2 data from the movementValue and stores in the vector2 variable you are creating called movementvector
  + Write private Rigidbody rb; This will create a private variable of the type rigidbody and call that variable “rb”,
  + The variable is private and not public because you don’t need the variable to be accessible from the inspector or other scripts right now
  + Next, inside the Start Function, write rb = GetComponent<RigidBody>(); This sets the value of the variable rb by getting a reference to the rigidbody component
  + Add new function called void FixedUpdate() below the start function
* Code for adding force
  + In the FixedUpdate function, add rb.AddForce(movementVector);
  + There are three vectors x, y and z
  + You need to create two new variables for individual input actions
  + Underneath the Rigidbody variable, add private float movementX; and
  + private float movementY;
  + In OnMove, add movementX = movementVector.x;
  + MovementY = movementVector.y;
  + Write in the FixedUpdate, Vector3 movement = new Vector3(movementX, 0.0f, movementY); The f signifies that this is a float value
  + Remove Vector from the rb.Addforce(movementVector);
* Code for movement speed
  + First, add a speed variable at the start, public float speed = 0;
  + Back in fixedupdate, add \* speed to the addForce function
* Moving the camera
  + Move the camera under the player in hierarchy so that the camera will move alongside the player
  + But the camera will also roll as the player roll
  + Write a script for the camera because the above method doesn’t work
  + Add public GameObject player; which will reference the player object
  + Add another variable declaration private Vector3 offset; this will store the offset value,
  + Take the current transformed position of the camera game object and subtract the transformed position of the player game object to find the difference
  + Add offset = transform.position - player.transform.position;
  + You will use that to camera game object’s transform position and this needs to happen every frame so write it in update function
  + transform.position = player.transform.position + offset;
  + Now, when the player moves, the frame before displaying the camera, the camera game object is moved into a new position aligned with the player game object before the frame displayed.
  + This works because only the position is added and not the rotation
  + However, you don’t control the order all of the update functions happen, that means the update can run before other scripts
  + The solution is to add LateUpdate which will run after all other scripts
  + You have to create a reference to the player game object \*\*\*\*
  + Drag the player component to the camera player tab
* Setting up the play field
  + Create an empty game object and name it walls
  + Create a cube named west wall and place it under walls
* Creating pick up objects
  + Create a cube, make it small, rotate it and color it
* Rotating the pickup object
  + Remove the start function from the template
  + The rotate values need to change every frame
  + There are two ways to transform a game object: translate (moves the game object by transform) and rotate (rotates the game object)
  + It has two possible parameters: Using a vector3 variable or 3 float variables xyz
  + Add transform.Rotate(new Vector3 (15, 30, 45) \* Time.deltaTime); make sure the “t” is lowercase because you are referring to the component not the variable type
  + Deltatime is perfect for smooth action because it’s a float that represents the differences in seconds between the last frame, multiplied by per second rather then per frame
* Making the pickup object into a prefab
  + A prefab is an asset that contains a template or a blueprint of a game object or a game object family
  + All the prefab objects will be updated with just one
  + Drag the pickup object from the hierarchy into the prefab folder in project
  + When you drag something from the hierarchy into the project window, unity creates a prefab asset containing a template or blueprint of the game object
  + The game object will turn blue when it becomes a prefab
  + Create an empty object and put the pickup object underneath it
* Detecting Collisions with collectibles
  + You will use the Ontrigger function on the player script
  + Write private void OnTriggerEnter(Collider other)
  + Other is the identifier that the sphere will hit
  + In the body, write other.gameObject.SetActive(false); this code will disable game objects correctly
  + Tags allow you to identify game objects by comparing tag values to a string
  + First, set a tag value for the pickup object
  + Add a tag called PickUp and apply it to the pick up object
  + Above the ontriggerenter, write if(other.gameObject.CompareTag(“PickUp”)) {}
  + Comparison is case sensitive \*\*\*\*
  + Paste the other code into the curly brackets
  + Then make the collider into trigger colliders Ex. A player can run into a door and a message will open that says you have discovered a new route or every time a player walks around a particular corner, spiders will drop from the ceiling
  + Any game object with a collider and a rigidbody is considered dynamic which means unity won’t treat it as static
  + If the pickup object is static, it will take unity longer each frame to calculate the physics and it would do it each frame because they are rotating
  + Add a rigidbody component to the pickup objects
  + The use gravity checkbox can prevent them from going down
  + Enable the ls kinematic (A kinematic rigidbody will not react to physic forces)
  + Standard rigidbodies are moved by physics forces and kinematic rigidbodies are moved by transform
* Adding score
  + Add a variable for integer count – private int count;
  + Adding an initial amount for count variable – write under Start function count = 0;
  + On the OnTrigger function, count = count + 1;
  + User Interface and Text Mesh Pro
  + Add text mesh pro from hierarchy, the important thing is that all UI elements must be the child of the Canvas behaved correctly
  + Rectangular transform for the text, Hold shift and alt to get it in the left corner
  + Before coding, you need to add information about text mesh pro in the script
  + The details about the UI elements are held in the name space
  + Write a new line called, using TMPro;
  + Under the public variable called speed, create a new variable called
  + public TextMeshProUGUI countText;
  + Underneath the OnMove function, write void SetCountText() {}
  + Write countText.text = “Count:” + count.ToString();
  + Under start function, write SetCountText();
  + Inside the ontrigger function, write SetCountText();
  + Drag the counttext object into the player component
  + Go to the event system object and replace the shit
* Creating a Game end message
  + Create a new textmeshpro object and transform the text whatever
  + Now, you need to add a reference for the UI text in the script
  + With the other variables, write public GameObject winTextObject;
  + Set the starting state to be disabled
  + Under Start function, write winTextObject.SetActive(false);
  + Underneath the setcounttext function, write if(count >=12) {}
  + In the curly brackets, write winTextObject.SetActive(true);
  + Don’t forget to drag the gameobject into the player component

Citations:

<https://answers.unity.com/questions/1255990/how-to-change-the-color-of-an-object-when-it-colli.html>

https://answers.unity.com/questions/1427037/how-to-generate-a-random-color.html

**Needles to slay tutorial**

Variables

* When creating a variable, define the type of variable and name the variable.
* Types of variable
* Whole number = integer
* Line of text = string
* A semicolon defines an end of an instruction
* That one variable can have different assigned values within the same script, because the computer executes instructions sequentially
* <https://docs.unity3d.com/ScriptReference/index.html> Scripting reference
* Save state so that you can restore any changes after
* // is for comments

Functions

* Void AddingNumbers(float num1, float num2)
* Void means no type or empty because its not stored anywhere
* Adding Numbers is the name of the function
* Parentheses are required even if the function takes no variables
* Floats are fractional numbers, add an “f” after a number to make it float
* Function call has a function name, set of parentheses containing two values and semicolon
* Change the parameters in the function to substitute the variables
* The curly braces {} are define scope. That means the only variables declared inside the function exist there.

Classes

* A class enables you to group together related functions and variables.
* Subclasses have access to the features of the parent class
* Use dot operators (a full stop) to access different variables within a class.
* Class as a type
  + Vector3 myVector = new Vector3();
  + New means to create a new variable space in memory
  + It’s only when you want to create a new Vector3 variable that you need to use that new syntax!

int NextAngle()

{

int currentAngle = angle;

angle = Helpers.WrapAngle(angle + step);

return currentAngle;

}

The function is int and not void because not only is it going to do something, it will also return a result (the angle for the new potion).

The WrapAngle function is a function that we’ve written for you, which enables you to store an angle between 0° and 360°.

You could write a function called Init to initialize those values to parameters you give them. But classes have a very useful feature called the constructor**.**

In the SpawnerSample script, create a new line at the top of the Start function (above the first call to SpawnPotion).

LootAngle myLootAngle = new LootAngle();

The constructor is a special function that has exactly the same name as your class and no type (for example, void) written before it. The compiler will create a “hidden” constructor by default which does nothing. If you create your own, you can give it specific parameters.

LootAngle(int increment)

{

step = increment;

angle = 0;

}

Put this in the lootangle class. Now the constructor takes one parameter: the increase each time the NextAngle function is called.

Replace the instructions inside the start function like this

SpawnPotion(myLootAngle.NextAngle());

There would be errors for the function LootAngle(int) which is inaccessible due to its protection level.

**Public and Private classes**

Private classes can only be accessed by the class itself or its subclasses.

Making a class public exposes it to the Unity Editor so it can be changed in Unity.

Add a public beside LootAngle(int increment) function and int NextAngle() function.

MonoBehaviour is the base class from which every Unity script derives.

When you use C#, you must explicitly derive from MonoBehaviour.

**Derivation and Inheritance**

An enemy class group together anything related to enemies.

The derived classes FlyingEnemy and WalkingEnemy will automatically inherit (be able to access) all the functions and variables of the Enemy class. It will look like this

public class Enemy

{

public void Attack()

{

}

}

public class FlyingEnemy : Enemy

{

}

The colon identifies **FlyingEnemy** as derived from **Enemy**.

You could then write code which accesses the function **Attack** inside the **Enemy** class without redeclaring the function.

Your scripts in Unity derive from the base class **MonoBehaviour**. This means that the script can access a range of variables and functions created within the base class that every Unity user can use such as GetComponent, transform.

public override bool Use(CharacterData user)

Use is a public bool function. The bool variable type stores one of the two Boolean values: true or false.

There’s an unfamiliar keyword in its declaration: **override**. As its name suggests, the keyword is used to override the same function in the base class (UsableItem.UsageEffect). This enables you to write generic code that will work for any Use function.

When an object is used, the function is called and it returns a bool value and if it has been used, it will be removed from the inventory. If the health is full, the function will return false and the potion will not get used.

Getting the Character Control class for changing speed

* GameObject = user.gameObject;
* CharacterControl control = gameObject.GetComponent<CharacterControl>();
* Control.ChangeSpeed =

**Late Night at Jennings Project**

The GameObject, John Lemon, has a component called a **Skinned Mesh Renderer**. This is what enables you to see the character.

the Animator component on the character’s parent GameObject will change the rotation of all the bone GameObject Transform components.

In Unity, models work like read-only Prefabs. They’re blueprints for creating instances of that model, but the blueprint itself cannot be changed.

The first property is called **Controller**. Animator Controllers contain a **state machine** which determines what animation the Animator component should be setting for its hierarchy at any given time.

Animator Controllers have Layers and Parameters

* Parameters- The Animator Controller’s State machine makes decisions based on the current values of its Animator parameters.
* You will need one parameter for every independent variable which can affect the animation that the character is playing.
* Four types of parameters: float (a number with a decimal place), int, bool, trigger (doesn’t hold a value, this causes a change from one animation to another)

Animating

* Drag the animation from into the editor
* The default state is displayed in **orange**. In this case, the default state is **Idle** because you dragged it in first.
* In order to add some logic, you need to make **Animator Transitions**.
* Your character needs to be able to change back to idle from walking, so repeat this process to create a transition from Walk to Idle.
* If Has Exit Time is true (the checkbox is enabled) then after a certain amount of time has passed the transition will automatically be taken and the state machine will play the next state.
* Provide a reason to make transition with a **Condition**
* The transition from Idle to Walk is required when the character is walking — that is, when **IsWalking** is **true**.
* Select the JohnLemon GameObject in the Hierarchy, then drag the Animator Controller to the Controller property of its Animator component in the Inspector.
* A **Rigidbody** component marks a GameObject as something that is part of the physics system that can move.
* Apply Root Motion is enabled on your Animator component, so any movement of the root in the animation will be applied every frame.
* So why does the JohnLemon GameObject move at all? This is due to the **Update Mode** of the Animator.
* An Animator component can change when it performs its Update. By default it performs this in line with rendering. This means that the Animator is moving the character in Update and the Rigidbody is simultaneously moving the character in Fixed Update. In the Update Mode property drop-down, select **Animate Physics**.
* Add a capsule collider that fits the character so that it will move according to physics

**Scripts for animating**

* MonoBehaviours are special types of scripts that can be attached to GameObjects just as components can.
* In C# it doesn’t matter in which order methods are declared in a class, but the order in which methods perform their operations matters a lot.
* float horizontal = Input.GetAxis ("Horizontal");
* Create a new float variable and call it horizontal; set that variable equal to the result of this method call
* GetAxis needs the name of the axis it’s trying to find the value of, in this case is the horizontal axis
* The data type for this method is a string or text. Using “” marks it as a string
* The **scope** of a variable is the area of code where it can be used, for example since the float variables are declared in the Update function, it can only be used in that function.
* Add Vector3 m\_Movement; above the method definitions.
* **Naming Conventions** are used to identify a particular object or class object
* In the Update, add m\_Movement.Set(horizontal, 0f, vertical);
* The movement vector is made up of two numbers that can have a maximum value of 1.
* Normalizing a vector means keeping the vector’s direction the same, but changing its magnitude to 1. So, add m\_Movement.Normalize ();
* Every frame, the computer should be doing these things
  + Telling the Animator component whether or not the character is walking,

bool hasHorizontalInput = !Mathf.Approximately (horizontal, 0f);

You are creating a **bool variable** called hasHorizontalInput. Then you’re setting that equal to the return value of a method. This method is called Approximately and is from the Mathf class. It takes two float parameters and returns a bool.

The exclamation mark is called the logical negation operator and it inverts a bool, setting true to false and false to true.

bool isWalking = hasHorizontalInput || hasVerticalInput;

This line creates a new bool variable called isWalking. The two vertical lines are the **logical or operator**. This compares the bool on each side. If either of them or both of them are true then it equates to true, otherwise it equates to false.

* + Declare the animator variable Animator m\_Animator;
  + Set up reference by m\_Animator = GetComponent<Animator>(); in Start
  + GetComponent is something that you already have access to: it is part of MonoBehaviour so you don’t need to declare a class. The angle brackets <> are because the method is **generic** method with **type** parameters.
  + m\_Animator.SetBool ("IsWalking", isWalking); The first parameter is the name of the Animator Parameter that you want to set the value of, and the second is the value you want to set it to
* **Rotation**
  + Set a turnspeed variable so that the character can turn and write
  + Vector3 desiredForward = Vector3.RotateTowards (transform.forward, m\_Movement, turnSpeed \* Time.deltaTime, 0f);
  + It sets it to the return of a method called **RotateTowards, has four parameters:** the first two are Vector3s, and are the vectors that are being rotated from and towards respectively. The next two parameters are the amount of change between the starting vector and the target vector: first the change in angle (in radians) and then the change in magnitude.
  + Quaternion m\_Rotation = Quaternion.identity;
  + **Quaternions** are a way of storing rotations
  + m\_Rotation = Quaternion.LookRotation (desiredForward);
  + This line simply calls the **LookRotation** method and creates a rotation looking in the direction of the given parameter.
* **Apply the movement and rotation to the character**
  + Add reference to the Rigidbody through Rigidbody m\_Rigidbody;
  + After that, add m\_Rigidbody = GetComponet<Rigidbody> ();
  + What you actually need is some of the root motion of the animation but not all of it — specifically, you need to apply the movement but not the rotation. Below the Update method, declare a new method:
  + Void OnAnimatorMove () {} This method allows you to apply root motion as you want, which means that movement and rotation can be applied separately.
  + Add m\_Rigidbody.MovePosition (m\_Rigidbody.position + m\_Movement \* m\_Animator.deltaPosition.magnitude); under the new method
  + New position starts off at the Rigidbody’s current position, and then the movement vector multiplied by the magnitude of the Animator’s deltaPosition. The Animator’s **deltaPosition** is the change in position due to root motion that would have been applied to this frame.
  + Apply rotation with m\_Rigidbody.MoveRotation (m\_Rotation);
  + Change the update to fixedupdate because OnAnimatorMove is actually going to be called in time with physics, and not with rendering like your Update method.
  + Add the player component script to the prefab

**Adding Environment**

* Go to prefab and add level
* The Directional Light in your Scene is set to the **Directional Type**. Directional Type lights mimic very distant light objects, such as the Sun.
* Indirect lighting is the additional lighting that occurs when direct light bounces off surfaces.
* **Global Illumination Lightmapping** feature simulates the bouncing of light within the Scene and writes (or ‘bakes’) it to an Asset stored within the Project.
* Disable the skybox and change the environment lighting to gradient
* The **NavMesh** is an invisible shape over the ground that defines an area within which selected GameObjects can move.
* Change the level prefab to static but disable ceiling pane from static
* The process of creating a Nav Mesh is called **baking**. **Window > AI > Navigation**
* The first settings refer to the agents that will traverse (the ghosts that will move around) the NavMesh — the **NavMesh Agents**.

**Camera**

* When you make a game, you have a few options to make sure that the camera followers the player character. One solution would be to write a script for this. However, Unity has a built-in solution to the problem: **Cinemachine**.
* One or more ‘virtual’ cameras are created in a Scene. These virtual cameras are managed by a component called the Cinemachine Brain. The Cinemachine Brain is attached to the same GameObject as a Camera component — by default this will be the Main Camera GameObject. The Cinemachine Brain manages all of the virtual cameras and decides which virtual camera (or combination of virtual cameras) the actual camera should follow.
* In the **Aim** section, change the drop-down menu at the top right from **Composer** to **Do Nothing**.
* change the **Follow** setting to reference JohnLemon’s Transform.
* In the **Body** section, change the drop-down at the top right of the section from **Transposer** to **Framing Transposer.** set the **Rotation** around the x-axis to **45**.
* Post-processing involves applying filters and effects to the game image before it’s rendered on screen
* Create a new layer called PostProcessingVolumes and add Post Process Layer component to the main camera
* Aliasing is when the edge of an object looks jagged and the pixel outline can be seen
* In the Post Process Layer component, change the **Mode** property dropdown from No Anti-aliasing to **Fast Approximate Anti-aliasing (FXAA).**
* Enable the **Fast Mode** checkbox below the dropdown.
* Create an empty game object called GlobalPost and set the layer to PostProcessingVolumes
* Add a Post Process Volume Component to GlobalPost.
* Enable the **Is Global** checkbox. Enable the **Mode** checkbox in the **Tonemapping** subsection. Using the drop-down menu, change its property from None to **ACES**.