Documentation Unreal Engine 5 C++ The Ultimate Game Developer Course - Udemy

5-15-23

1. Getting Started

Notes:

- View resources, GitHub for the project.

- Put this on YOUR GitHub.

- Samples tab: Contains starter games, code, assets, and learn from the examples.

Lyra and valley of the ancients

- Marketplace tab: Tons of free assets. Go in once a month and see "Free for the month".

Takes up no space until it is used in the project.

- In the launcher, click games, this allows us to create a games project in the engine.

- Blank blueprint project, add C++ later.

- Enable starter content, as well as Raytracing for higher performance.

In the editor

- Tooltips: Hover over something in the interface, it will tell you what it does.

- Shift + F1: Mouse to appear in viewport during a play session.

- Shift + Esc: Cancel the play session.

- Content Drawer: Ctrl + Space, dock in layout.

- Different editors, level editor, viewports.

- Left click and drag: Parallel movement, panning left and right.

- Right click and move: Looking up, down, left, and right.

- Left and right click at the same time: Moving around in the plane of the screen. UDLR.

- Left and Right click, WASD.

- Left and Right click, C, zoom in. Z, zoom out.

- Camera speed: Choose how fast to go with the camera.

View Modes

- Show button: Different flags and settings to toggle the view.

- Static meshes for example, uncheck it and we won't see them.

- Collision: Shows the collision detection for objects in the scene.

- Lit button, different viewpoints.

Unlit, good for dark areas.

Wireframe: Raw geometry in the scene.

- Perspective: Changes the view from perspective to orthographic or different angles.

- 4-Panel button at the top right, splits the scene into 4 windows.

- Hamburger Button: Displays shortcuts, FPS, Stats, bookmarks.

- Game View: Removes gizmos and shows what the game will look like, press G.

- Immersive Mode: F11 is fullscreen, press G to remove objects again.

- Screenshots: High Resolution Screenshot, saves to a screenshot folder in the project.

Object Manipulation

- Axis: Blue: Z, up and down, Red: X, Green: Y. Directional gizmos.

- Select Objects: Mouse icon, press Q. Selecting objects.

- Select and Translate: Directional icon, W, move icons around in the world and select.

- Rotation: Circle icon, E, rotate objects along axis.

- Scale: Arrow icon, R, scale objects along axis.

- Grid snapping: Change snapping increments.

- Rotation increments: Alter degrees.

- Local and Global Directions: Point to world space or object space.

- Snap surfacing: Drag in a new mesh, alter the way objects snap to surfaces.

- Duplicate objects: Alt and drag an object.

Panels

- The outliner: Contains a list of all the objects in the world. Create folders to stay organized.

- Search in the outliner for particular items

- Ambient sound.

Section Challenge

- Drag in some props from the starter content.

- Manipulate objects

- Translate, Rotate, Scale.

- Duplicate

- Snapping on/off.

- Add a new Bookmark.

- Enter game view, G and immersive mode f11.

- High Res screenshot.

- Check out the other built in levels, minimal default for example.

End of Section 1: Absolute basics

- Installed

- Level editor viewport

- Maneuver through the world

- Change view Modes

- Manipulate objects

- Panels

5-17-2023

Section 2: Realistic Landscapes

Quickly add to project:   
- Quixel Bridge – Tons of assets, Quixel Megascans

* “3D Assets” panel
* Alt + click, moves in the static mesh editor.
* Static Mesh
* Material
* Texture

Large Worlds:

* UE5 has introduced a new open world system, has one large map instead of multiple smaller ones. Loads one section that the player is in, world partitioning.
* Make a new map folder.
* World partition shows the world broken up into cells.

Lighting and atmosphere:

* We need 5 things:
  + Sky Atmosphere – Designed to create an earth-like atmosphere, scattering light in a realistic way. Up to two atmospheric lights are allowed, can have one for the sun and another for the moon.
  + Directional Light – Simulates a light source that is infinitely far away. Shadows cast from it are parallel. Simulates the atmospheric light source (Sun or Moon).
  + Sky Light – Captures distant parts of the level, applies captured light information to the scene as reflections, global illumination. Captures in certain conditions (Static: Updates whjen building lighting. Stationary and movable: Updates once on load and on “Capture”. Constantly when real-time capture is enabled).
  + Exponential Height Fog – Simulates fog, gets thicker the lower you go, two colors, one for the hemisphere of a planet facing the sun, and the other hemisphere.
  + Volumetric Clouds – New way of doing clouds they are more dynamic, 3-dimensional, and material driven. Scatter light like the atmosphere does.
* To make a default map when you open the project, use edit, project settings, maps and modes.
* Start with the atmosphere.
* Place actors panel gives a variety of common actors to place in the scene.
* Two suns: Directional lights, atmosphere sun light index. Ctrl – L to alter the position of a directional light. Ctrl – Shift – L to move the second sun. Temperature value, affects color of suns.
* Gimble lock.
* Mobility settings for a light.
  + Static – Lighting cannot be changed in game; they are the fastest kinds of light to produce. Allow baked lighting, bake it into the scene because it will not change the value.
  + Stationary – Color and intensity can change in-game. Allows partially baked lighting.
  + Movable – Can be moved and changed in-game, dynamic shadows. Very expensive to produce and used the most sparingly. Day and night cycle along with the skylight.

Landscape:

* Under select mode, this gives us many options to choose from, this time we will select “Landscape” mode.
  + Generates a massive template which we can sculpt and manipulate.
  + Number of components determines the size of landscape in its grid-like orientation. 8x8 default.
  + Once generated, it opens a set of modes for you to edit your terrain, such as sculpt.
  + Shift 1 is selection mode, shift 2 goes back to landscape mode.
  + Select individual portions of the grid, LandscapeStreamingProxy.
  + Use objects to gauge the scale of your world. 0,0,0 for the world origin.
  + End key to snap to the floor or current surface.
  + Center your components such as the sky devices at 0,0, additionally, make use of the folder to organize yourself.
  + Want rolling hills, outdoors level, we need to sculpt.
* Brush falloff - higher makes a more rounded edge.
* Round Brush – Default circular brush.
* Alpha brush – Uses a mask image as the base, i.e. a checkerboard pattern.
* Pattern brush – Similar to alpha, uses the pattern as the base.
* Landscape brush – For altering entire squares of the landscape.
* Visualize your map, zoom all the way out.
* Erosion, getting a nice rough look to the overworld.]

Landscape Material

* Materials dropdown menu -> Browse to Material -> Content drawer with the materials
* Material and texture editors, bouncing light off the pixels to give a more 3D effect to the material.
* Get materials from quixel bridge.
* Base color and normal textures.
* Prefix M\_ to name materials.
  + When making a new material, we are going to configure the base color and the normal.
  + “Fully rough” No shine or glossy textures for the landscape.
  + LAYERS
  + Add to the layers array, layer blend.
  + Name layers in the layer blend for each material that you are working with.
  + Once all base colors and normals are mapped to each other, you can assign the landscape material.
* Open the “Paint mode” in landscape mode, this will have all our layers that we mapped to our materials.
  + “Create layer info” – Weight blended will stack on top of another and will blend. Non weight is not blended.
  + Save to the same folder as your landscape material.
  + If the world does not load, use world partition and select the cells to load.
  + Painting went, okay? Blending helps.

Foliage Painting

* Importing assets from Quixel Bridge and Unreal Marketplace
* Instancing – The mesh and data are the same, locations in memory are identical and the only difference is the orientation of them in the world.

Packed Level Actors

* Allows us to pack a level instance into an actor, such as combining multiple parts together into one actor to use between levels.
* Save the level instance in a new folder, level instances.
* Drag into levels as a whole group of meshes.
* Can refine packed level actors by opening them in their own level, this allows for finer editing.

Valley of the Ancient

* Super high-fidelity project created by Epic Games, we can take geometry, assets, and even the character Echo to use in our own project.
* Migrating folders, always put it in Content of the project you are trying to migrate to.
* Right click on a folder or asset, click migrate.

Level Instancing

* Creating an instance allows us to bring in whole levels once they are assigned to the instance.
* Very useful for importing entire levels, like dungeons in this scenario.
* Select the closest version for assets that do not have the version you are using.
* To copy a level, click duplicate. This will allow you to modify a map, but if you do not like it, you can go back to the original.
* Breaking level instances splits it back into the original actors.

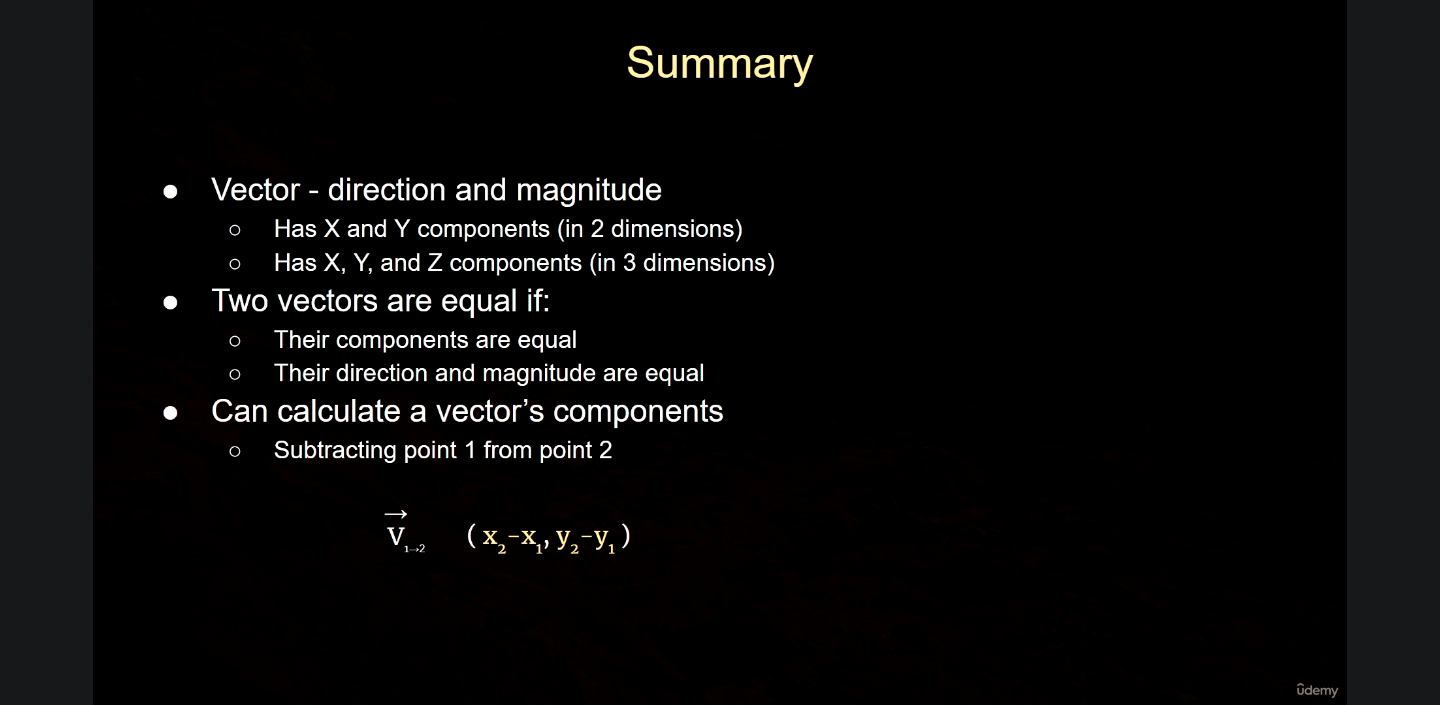
Section 3 – Vectors, Rotators, and Trigonometry

Coordinates (Representing location in a 3D space)

* X: Forward and backward forever, tick marks (+, -) increasing and decreasing respectively.
  + Your X-coordinate can be represented with a scalar, a numerical quantity or distance from the 0 point on the X-axis.
* Y: Y Scalar is the distance from 0 on the Y axis, up and down indefinitely, we are now in 2D. The XY coordinates show where you are, this is called an ordered pair.
* XY plane: The set of all XY coordinates you can move on a plane.
* Z: Up and down in the 3D space, XYZ coordinates, ordered triples, the origin point is (0,0,0).
* Summary:
  + Scalar – A single numerical value
  + X coordinate – position along the X axis
  + XY coordinate – position on the XY plane, represented by an ordered pair.
  + XYZ coordinates – position on the 3-dimensional space, represented by an ordered triple.
  + (0,0,0) is the world origin for 3D. (0,0) is the origin in 2D spaces.

Vectors

* An arrow with a starting point and an ending point is called a vector.
  + If we move from the point (5,5) to (20,20) on the XY plane, this is a vector.
  + A vector contains a direction and a magnitude. “Tail is at 5,5 the head is at 20,20”
  + Magnitude means the size or extent, which is represented by length.
  + We store the information of a 2D vector with XY components.
    - In the example, our X component is 15 and the same for our Y, this is because we must travel 15 units across the X and 15 units up the Y to reach the head of our vector.
  + Mathematically, this vector (V) can be shown as V = (20 – 5, 20 – 5) = (15, 15) components
  + The formula for this example could be, V = ( x\_2 – x\_1, y\_2 – y\_1), subtract the coordinates of x2 and x1, and vice versa.
  + If a vector has the same XY component, they are equal making them the same vector. A vector is only defined by its direction and magnitude, not location. V1 = V2.

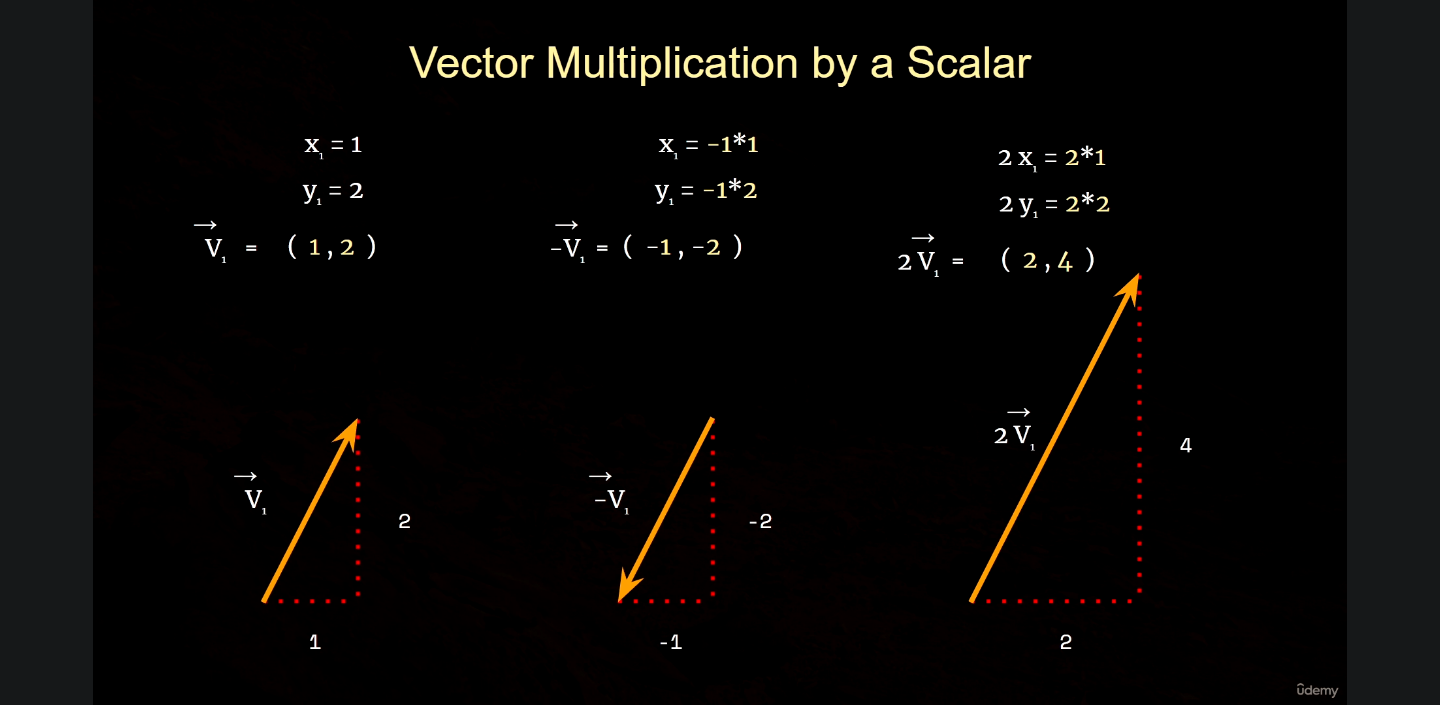


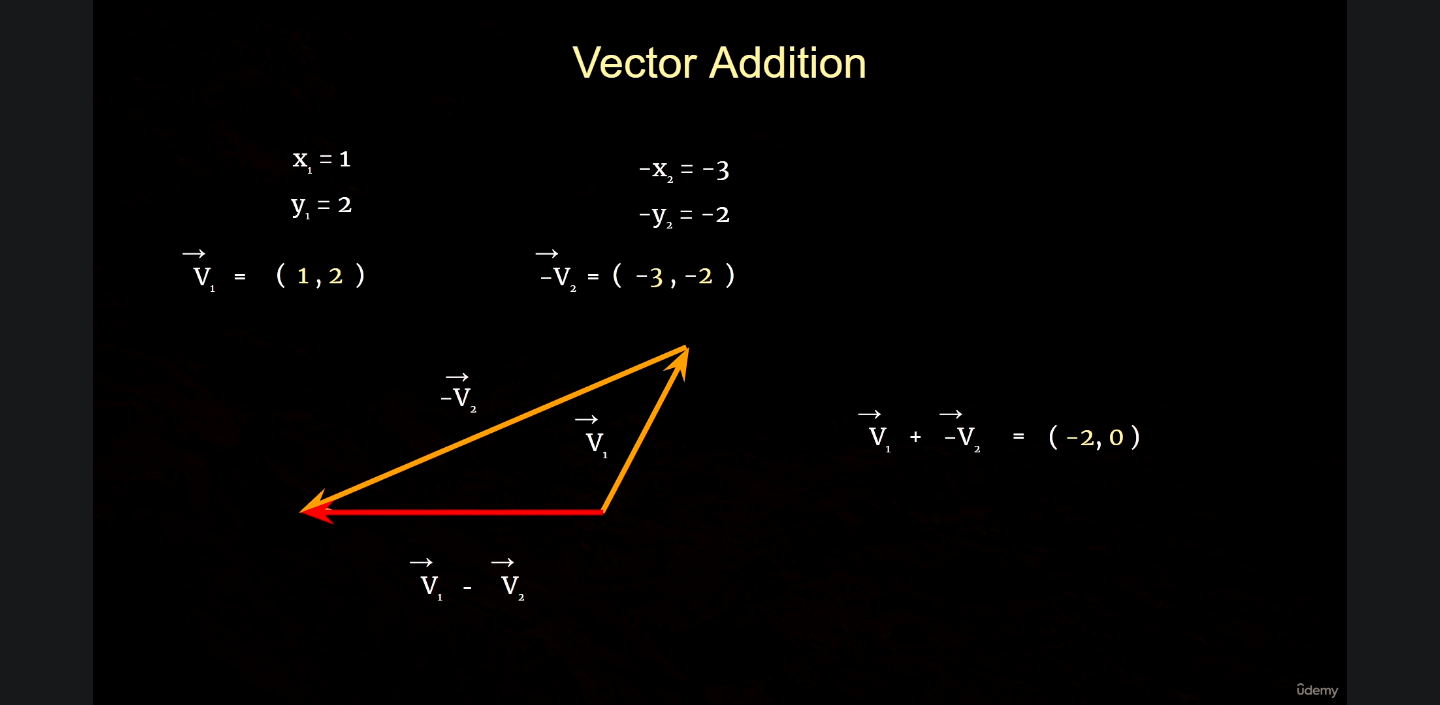
Vector Examples

* 2D: If our character has a set of coordinates (4, 4) and we have an enemy’s location with coordinates (2, -4). We need a vector from the enemy to the character, an AI for example would need pathing to go towards the character.
* Enemy is point 1, and character is point 2, we can calculate the vector required.
  + V 1 -> 2 = (4 – 2, 4 – (-4)), V 1-> 2 = (2, 8).
* From character to enemy, V 2 -> 1 (2 – 4, -4 – 4) -> V 2 -> 1 (-2, -8).
* It is the inverse, for character to enemy and the other way around, they are opposites. When we have opposite vectors, then the signs are the opposite.
* 3D: We have a sniper (1) at the coordinates (0, 0, 7) that is aiming down at a character (2) on the ground at the coordinates (4, 3, 0). The sniper needs to know the vector from his position down to the character’s.
* Calculate the Z, which is the same as the previous formula:
  + V 1 -> 2 = (X2 – X1 – Y2 – Y1 – Z2 – Z1) -> V 1 -> 2 = (4, 3, - 7).
* We need to travel 4 x units, 3 y units, and 7 units in the negative Z direction. The vector formula checks out.
* If the character wants to shoot back at the sniper, then V 2 -> 1 = (-4, -3, 7).

Vector Operations

* Vector Multiplication by a Scalar.
* Negative V1 will have an X component of negative one times the V1 component.
* Magnitude is scaled by 2, this is going through each component and scaling it up.



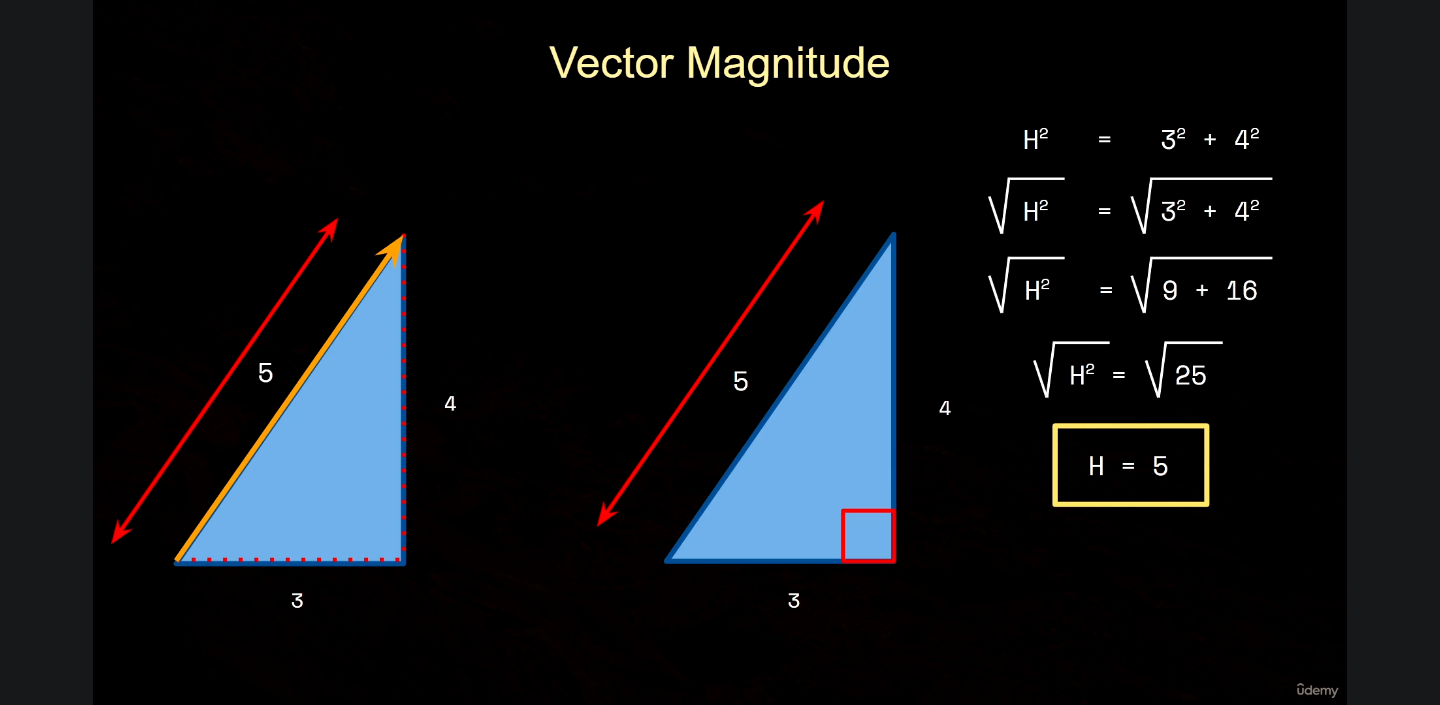
* Dividing a vector by a scalar.
* V1, component (2, 4), x1 = 2, y1 = 4.
* We can multiply by one half, this is ½ \* V1, which is the component (1,2). This is half the length of the original vector, or half the magnitude.
* Vector addition.
* V1 = (1, 2), V2 = (3, 2). If we want to add these vectors, we take the x components and y components and add them together. This will be V1 + V2 = (1 + 3, 2 + 2) (X, Y).
* The formula for adding vectors is V1 + V2 = (X1 + X2, Y1 + Y2). This is called component wise addition.
* Vector subtraction is very much the same.
* V1 = (1, 2), V2 = (3, 2). V1 – V2 = (1 – 3, 2 – 2). V1 – V2 = (X1 + X2, Y1 + Y2). Component wise subtraction.
* Subtraction:
* 
* Component = (-2, 0)
* Addition:
* A picture containing screenshot, line, text, design

  Description automatically generated
* Component = (4,4)

Vector Operations Examples

* 2D: Character has position (1,1) and is aiming at an enemy; we use the vector describing the path of the character’s projectile. V1 = (2, 1), if we want to increase the range of the character’s projectile to hit the enemy by 2, we can use a scalar of 2 to multiply the new vector.
* 2 V1 = (2 \* 2, 2 \* 1) -> 2 V1 = (4, 2). By multiplying the vector V1 by 2, we get the new vector that allows the projectile to reach the enemy.
* 3D: Character (2, 2, 0), these are the 3D cords for our character. Our character has a vector that starts at their coords, for this example, it’s an impact point of a projectile. To get there, we would need to have the components (3, 0, 2) and our character’s coordinates.
* Ve = (3, 0, 2).
* Whenever we have 3D coordinates, we can represent it with a vector that starts from the world origin and ends to the 3D location. These are called “location vectors”.
* Now that we have 2 vectors, we can calculate the vector that starts at the tail of the first and the head of the second vector.
* To calculate the third vector, we simply add the original two together.
* Vc + Ve = (2 + 3, 2 + 0, 0 + 2) = (5, 2, 2). This is used to calculate the point in space, we can use two vectors and their operations to find these points.

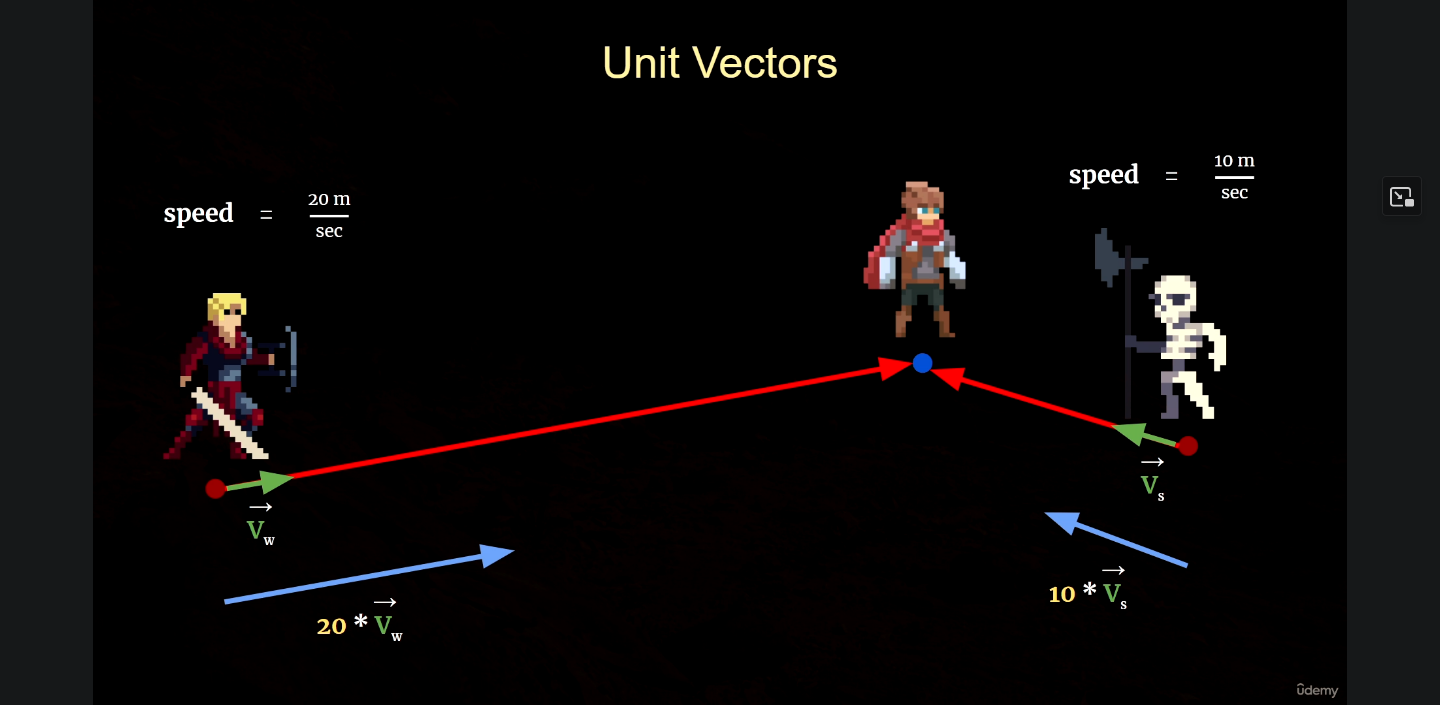
Vector Magnitude

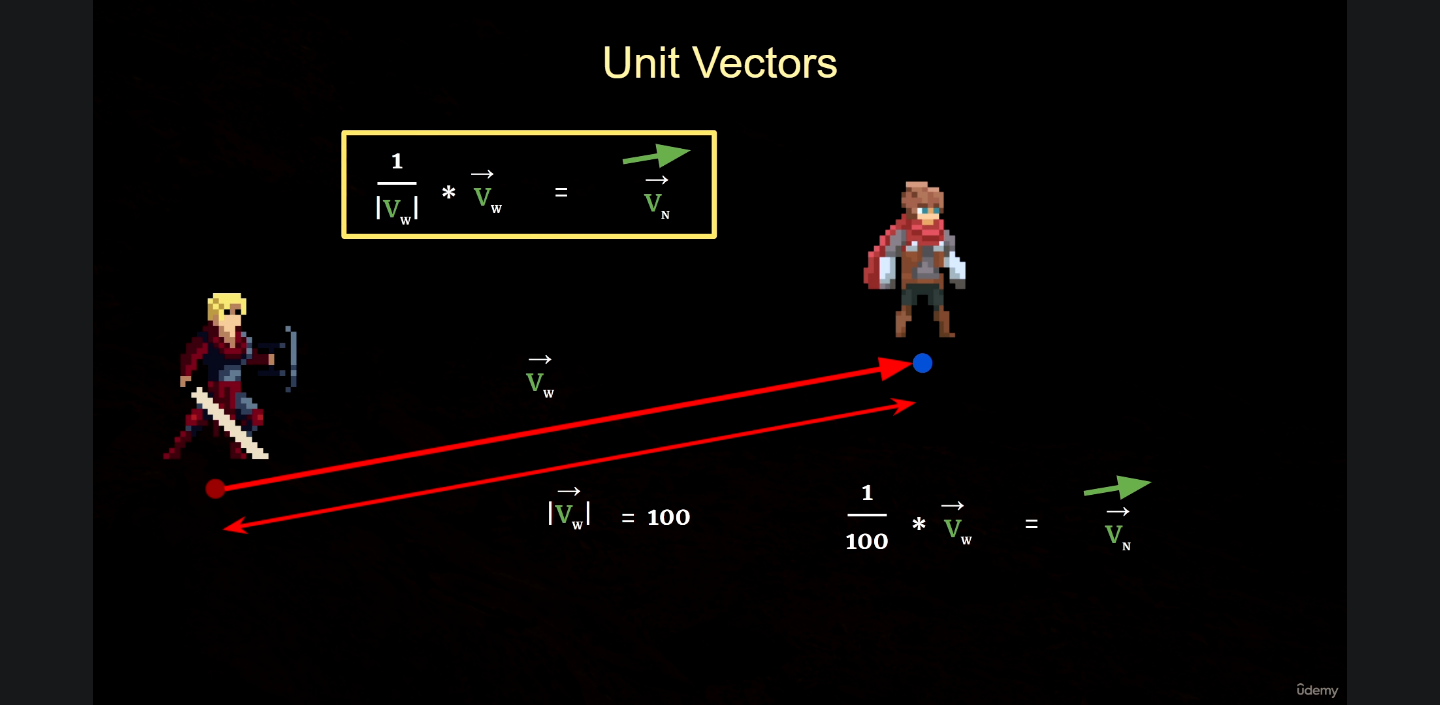
* We can calculate the magnitude of a vector by visualizing them as a right triangle, all vectors form a right triangle. We have two legs and a hypotenuse.
* The parts of each right triangle are connected via the Pythagorean Theorem.
* 
* The Hypotenuse would be the length of the vector, or its magnitude.
* A picture containing text, screenshot, line

  Description automatically generated
* Math libraries in programs are made to simplify these calculations for developers, the logic is there but we do not need to consistently calculate all these functions and equations by hand.

Vector Normalization

* The Pythagorean theorem works with 3D vectors as well.
* Calculating magnitude with the Pythagorean theorem both for 2D and 3D vectors.
* Unit Vectors
  + Three vectors with a length/mag. of 1, any vector like this is a “Unit Vector”.
  + Vectors should be smaller or larger based on movement speed of an enemy per se.
  + Each vector needs to be scaled, but it will point towards the character.
  + This is where we use unit vectors, each enemy has a unit vector that serves as a base, which is then multiplied by the desired movement speed for our particular enemy.



* Reducing a vector down to one, but still pointing in the same direction towards the character. This is called “normalization”, with the resulting vector that is the normalized vector.
* 

Rotators

* Left hand v Right hand coordinate systems.
  + 
  + Right hand is the most applicable, it has both X and Y on the ground, but Unreal uses the left-hand coordinate system.
  + Local and world axis. X is considered the “Forward vector”, the Y is considered the “Right vector”, the Z is the “Up vector”.
* If we tilt the object by 45 degrees, the X and Z change, but the Y remains the same, we changed the pitch of the object. This is the rotation of the Y axis.
* The X and Y can be tilted, but the Z hasn’t changed, 45 degrees about the Z axis is known as Yaw.
* Y and Z are changed, X is the same, this is known as roll, which is rotation about the X axis.
* Every object has a pitch, yaw, and roll when it is rotated. When aligned perfectly with the world axis, these values are all zero.
* A rotator has pitch, yaw, and roll components, which are scalar values for rotations. A rotator in UE stores the rotation in degrees.
* Challenge: A character in a rogue lite video game has an ability that allows them to teleport over to an enemy for a deadly attack. In a 3D space, our character is at the position (5, 0, 0) with the enemy at position (0, 0, 7), on a platform above them. We would have to use vector calculations to teleport up…I don’t exactly get it, but I took good notes so let’s refer to them if needed.

Section 4 – C++ in UE

Choosing an IDE

* Some good IDEs for UE are Visual Studio, Xcode, and Visual Studio Code.
* Recommended with a Windows machine and Visual Studio.
* Rider for Unreal Engine by JetBrains and Visual Assist X by Whole Tomato. Not free, but they are great.
* For VS: Disable the entire error list and showing all external dependencies folders, enable intellisense, show inactive blocks.
* Macros\*
* Live coding > Hot reloading.
* Visual Studio should be configured.

C++ Refreshers

* Classes and Inheritance, two main principles for OOP.
* Learn C++ for Game Development course.
* A screen shot of a computer

  Description automatically generated with low confidence
* Parent class can have a child class, and more child classes after that. This is known as an Inheritance chain.
* There are many children that derive from the same parent class.
* Multiple Inheritance: The child class can inherit functions from two parent function.
* Pointer: A variable designed to hold an object’s address.
* The object pointed to that will determine which one is called in the program.
* A screen shot of a computer

  Description automatically generated with low confidence
* UE has its own inheritance hierarchy.
* UObject -> AActors -> APawn -> ACharacter. The A stands for actor, which is what it is derived from. The pawn and character classes are inherited from the actor class..
* UObject: Stores dats, cannot be placed in the level.
* AActor: Can be placed in the level, have a visual representation like a mesh.
* APawn: Can be possessed by a controller, which is an actor designed for controlling pawns. Receives the information, processes, and moves the character via the input.
* ACharacter: Can also be possessed by a controller, character movement component which has more sophisticated movement than a pawn. Character-specific functionality.

A screenshot of a computer

Description automatically generated

* Is A vs Has A relationships.
  + A screen shot of a computer

    Description automatically generated with low confidence
  + This is what is known as a “Is A” relationship.
  + Outer class, inner class. The inner class can have its own variables, and this nested pattern is repeated throughout UE.
  + Package -> World -> Level -> Actor -> Components.

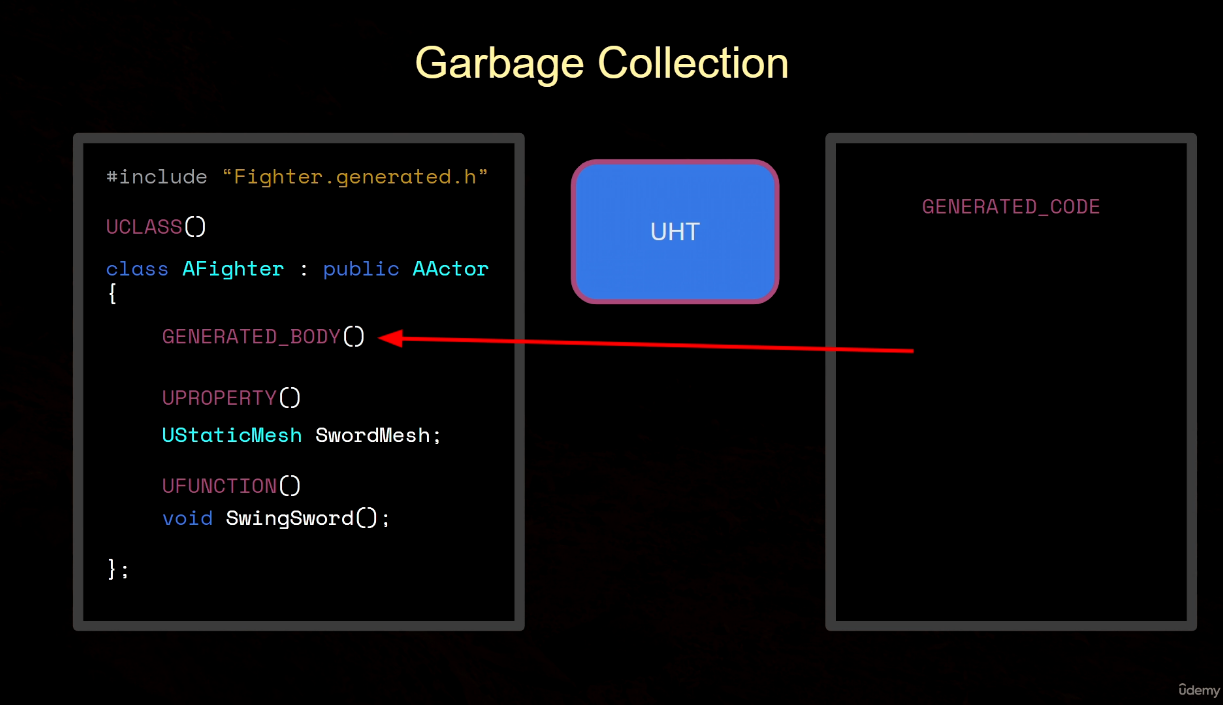
A screenshot of a computer screen

Description automatically generated with low confidence

* Meshes, textures, sound files, and others are nested within the package, but they are not Sub-objects.
* Keep these concepts in mind as we develop games in UE.

Reflection and Garbage Collection

Reflection: The ability of a program to analyze itself at runtime, harvests data about itself.

* UE has its own reflection system, which is responsible for merging data with the UE System, exposing data to blueprints, and managing garbage collection.
* Garbage collection: When a program manages memory associated with the project and automatically deletes objects when they aren’t being used, freeing up memory.
* If an object is in the garbage collection system, when the object is no longer being referenced, the object is deleted. C++ makes you do this yourself, which can result in memory leaks.
* Special Macros() – A macro is a given fragment of code that has been assigned a name. Anywhere that macro is called, that is where the code is placed by the preprocessor, and by the time compilation starts, that code will be there in place of the macro’s name.
* For code participating in the garbage collection/reflection system, UE demarcates them with the macro UCLASS, meaning we want it to be apart of those systems.
* This allows us to expose variables to blueprints.
* Classes using reflection will have a special include at the top of the file, this is the class name generated.h file, which has auto generated code which UE generates by seeing the macros.
* Generated body macro, auto generated by the UHT for the header class. Some macros have parentheses which take inputs, called specifiers, which alter the way that macros behave or are exposed to the reflection and blueprint system.
* 
* UHT harvests this information during runtime.
* UCLASS macro, variables are marked with the UProperty(), functions are marked with the UFunction().

Section 5 – The Actor Class

Actor Creation

* Creating our first class, which for Unreal will be an actor, based on the hierarchy.
* While we have built a blueprint project, we can convert it to a C++ project very easily.
* Empty would have a default constructor and destructor, but we can choose the engine classes as well.
* Common classes, all classes.
* Can place the class in a separate folder for instance for organization, if the class were for an item, we could call it item.
* Public or private
  + If we choose private, we get a new folder called private.
  + If public, we can separate our header and source files.
* C++ Classes folder -> Project Name -> Public -> Path name
* CoreMinimal functionality, this is for actors and their bare necessities.
* PROJECT\_NAME\_API macro, default generated by UE.
* Regular C++ class with beefed up functionality, linking the class with the reflection system, exposed to blueprints, etc.
* Tick() is called rapidly, once every frame.
* Super::FunctionName() – we are calling the parent name of the function.
  + Actor version may implement some functionality that we need at the start of the game.
* We now have our first class and work with the BeginPlay() and Tick() functions.

Blueprint Creation

* We can drag “Item” directly into the world because it derives from the actor class.
  + It will have no mesh or visualization.
* We do not drag in raw C++ classes, but instead derive blueprints from them.
  + Think of blueprints as children, inheriting the functionality of the original C++ class.
* We want to create a blueprint based on our C++ class; we can do this by:
  + Right clicking on the C++ class, then choosing “create blueprint class based on item.”
  + Go into the folder we want our blueprint in, right click and choose “Create Blueprint Class”.
* Prefix with BP when naming the asset.
* Opening the BP editor, we can see the local X Y Z of our actor, as well as a gizmo to get a closer look at its properties.
* Components panel: All actors have at least one component, the root component by default.
* Viewport, Construction Script, Event graph.
  + In the event graph, we can use blueprint nodes to create our game logic.
  + Print String node is a function, we can tell because it has an “F” next to its name.
  + Press play within a blueprint to get a separate window and fullscreen play mode.
  + Print String has various properties such as duration, text color, and print to screen. This is a great way to test if our function or event is being executed.
  + Begin play is like an event, Print string is like a C++ function.
* Event is the terminology for delegate.
* Construction script is executed before the game starts. Event graph is executed during gameplay.
* If you are too far from the actor you will not receive the message.
* Control – Shift – Space, intellisense prompts.
* TEXT() macro for a string literal, always use it.
* Epic Games C++ Coding Standards – Article.
* Semicolon is optional for the UE LOG
* HA Reloading.
* Virtual Textures support option.
* Called with no errors, use the coding node to see results and press play.
* Compiling in the editor lets us see changes quickly.
* Ctrl F5 launches unreal engine in VS.
* UE\_LOG is good to use for verification of functions.
* Both my BP function and C++ are being called successfully.

Onscreen Debug Messages

* Actors will not be called unless they are in the same level, if the output message is not being called, make sure it is in the world.
* Print String goes to the screen, along with Log string it goes to the output log.
* Newer messages show up on the top of the stack.
  + Key allows us to specify a value to the message. If both have the same key, the newer message overrides it.
  + If you have different keys, they will show up together.
* Tick() is being called every frame.
  + For the print string function, assigning a key allows it to replace the old value, in this case with tick, it won’t spam the screen every frame.
* Delta Seconds provides the time between frames.
* Implicit type conversion. Every single frame we are printing the delta value which is updating the time between frames.
* A screenshot of a computer

  Description automatically generated
* Check your pointers before you use them.
* AddOnScreenDebugMessage(Key, Time, Color, Message)
* This all works for me, add some flavor text to the screen or output log.
* Delta Seconds = Delta Time

Formatting Strings

* VA\_ARGS = Variable arguments
* Format specifiers must tell the macro where to put a different data type in the output log.
  + C++ Specifiers related to printf, F is a decimal floating point.
  + FString contains a static function that allows us to create and manipulate strings.
* UE has its own versions of C++ types.
* Peek definition, viewing engine files allows you to see how certain types work.
* %s for strings.
* Check documentation for more information
* Either way, this worked, and intelli sense is back online for VS it seems which is a great sign.

Drawing Debug Spheres

* Line Color is like the F color in C++
* If we did not specify coordinates, the debug sphere will appear at the origin.
* Get Actor Location (F, C++, BP), fetches the X Y Z coordinates of any actor.
* World is a subclass of the package, debug sphere needs to know what world we are in so we use GetWorld(), returns a UWorld pointer.
* Null pointer? Check it.
* FVector in C++, in blueprint it’s called Vector.
* Int32 data type for UE.
* Visualize things that may be invisible.
* #define allows us to create a macro, which we can use to define certain values.
  + Function macros
  + Can do an IF statement in one line in C++.
* Must define the macro symbol to use in another class.
  + #include “Project Name/Project Name.h”

Drawing Debug Lines

* Visualizing locations in our world.
* If we add a vector and we add another, the result is the start point of the first one and the end of the last one.
  + Origin to the tip of the point.
* Start at the actor location and end at the actor location plus the forward vector.
* So far BP seems a lot easier…
* UE measures in CM by default.
* Can visualize the direction of a vector and other actors in debug mode.
* CTRL B to compile the built project file.

Own Header Files

* Right click on the project name folder, create header file.
* Put macros in another header, then you can include them in the cpp file.

Section 6 – Moving Objects with Code

Moving Actors

* SetActorLocation()
* SetActorRotation()
* Actor World Offset: Useful if we want to change our actor’s location and rotation every frame to give a continuous movement.
* Making the actors move per frame is not ideal, as frames are not a constant value.
  + Instead, we can use our own movement speed float value and multiply it by delta time to make it frame rate independent.

Trig Functions

* Sine function, in trig, we have various functions such as sine, cosine, and tangent, but these all do different things in game development.
  + Pass in a value to sine, it will return a value “Sin of T”.
  + The side that is opposite the right triangle is the hypotenuse. Sine takes an angle as input.
  + Side opposite to theta, adjacent to theta.
  + Sine = Opposite/Hypotenuse.
  + Cos = Adjacent/Hypotenuse.
* Unit Circle has a radius of 1 unit and a diameter of 2.
* Circumference = 2 \* Pi \* 1 for a unit circle.
  + The line outward from the origin in the unit circle always has a length of one and creates a right triangle.
  + Use the trig functions to calculate the legnths of 2 sides of the triangle in relation to theta.
  + Sine of theta is equal to the opposite side / hypotenuse.
  + Y = sine(theta) for the unit circle.
* Radians: 360 = 2 \* pi \* rad, 180 = Pi radians
* When we want periodic behavior, we can use Sine and Cosine for sinusoidal movement.
  + It is almost as if it is a bobbing motion.

The Sine Function

* My Blueprint tab shows everything we have in the event graph now.
* Adding a variable gives us the data type, Boolean by default.
* Getters and setters for variables, we are adding delta seconds to our variable, Treat, and returning the set value.
  + Hold Ctrl and drag for a getter, alt for a setter.
* Treat is our runtime variable.
* The Sine function will bob our item object to move up and down.
  + We can tweak the sine function if we so choose.
* C++: Use a RunningTime float value and add it to DeltaTime.
* Amplitude is the wave value of our sin object.
* Create variables in our header files for certain values, private.

Exposing Variables to Blueprint

* Can manipulate variables directly in the details panel rather than hard coding values each time and compiling. We can also select an actor instance in the world without affecting the blueprint itself.
  + Instance in the world and the default blueprint editor.
* UPROPERTY() Macro – Exposes a variable to the UE reflection system, which would allow it to be exposed in blueprint, garbage collected, etc.
  + Takes a parameter, in this case EditDefaultOnly, it only works in the default blueprint though.
  + Exposing the next variable, this time with EditInstanceOnly.
  + Changing the value of a particular instance will only affect that instance, and not the default. This makes much more sense now.
  + EditAnywhere allows us to perform edits on both the default and the instances.

Visible but not Editable

* Exposing runtime to the blueprint editor, UPROPERTY.
  + VisibleDefaultsOnly, initialized but we cannot change it because we did VisibleDefaultsOnly. Not visible for instances.
  + VisibleInstancesOnly, same thing but makes it invisible for defaults.
  + VisibleAnywhere.

A screen shot of a computer screen

Description automatically generated with low confidence

Exposing Variables to the Event Graph

* Read or Write access when exposed to the event graph.
* Multiple specifiers for UPROPERTY in one line.
* Move a Read only to the Protected section, not private. This will allow it to compile correctly and be moved onto the blueprint.
* BlueprintReadWrite allows us to read and write to the variable in the blueprint.
* Damped oscillation.
* Category specifier (Category = “Name”) in UPROPERTY.
* Meta allowprivateaccess = “true”, exposing to the event graph when private.

Exposing Functions to the Event Graph

* Transformations of the sin() function.
* Are we planning on calling this function outside the class? If so, make it public, if not it does not need to be public.
* Do you see yourself calling this function called in a child class? Use protected.
* If it is the only class you are using, put it in private.
* VA\_ARGS: Variable number of arguments, as many as we want.
* BlueprintCallable, makes the function callable in blueprint.
* Blueprint Pure Functions: Do not change anything, they just calculate a value and return it.
  + Signified with a Green F in the blueprint search.
* Circular movement, we have the capability to call a function to blueprint and allow us to input mathematical logic, such as creating circular movement.

Template Functions – Using Types as Parameters

* We may want a function that can take inputs as parameters but also a type as an input.
* Template type parameters, type must support addition and division.
  + Int32, Float, FVector, we can find the average of all these data types.

Components – Adding Functionality to Actors

* Actors can have components, this gives them functionality they may not have.
  + This would include visualization in the world without drawing a debug sphere.

A screenshot of a computer game

Description automatically generated with medium confidence

“Say we have an actor called Weapon. Now a weapon might have components. For example, you might need a mesh, so we would have a component for that. That way we can see what our weapon looks like in the world. Now, when you swing the weapon, you may wish to know when the weapon hits something. So, you might place an invisible box component around the blade of the weapon mesh that we can use to detect overlapping events when we hit an enemy. So, adding these components to the weapon adds more functionality so that our weapon is functional in the game”.

* Default scene root is the default component for an actor. In C++ it’s called RootComponent.
* Scene components have a transform, which is a location, rotation, and scale. (FVector, FRotator, FVector). Can be attached to other components.
* Whenever we call GetActorLocation() we get the RootComponent Location through it’s transform information.
* Attachment:
  + Components panel in blueprint -> Name of the BP (Self) -> Default Scene Root.
  + Scene root is invisible but supports attachment, we can attach a new component to the root.
  + The scene with move with the root, keeping its relative distance constant.
* Static Mesh Component:
  + USceneComponent -> UStaticMeshComponent

A screenshot of a computer

Description automatically generated with medium confidence

* If we create a new actor called a weapon, it will have the default scene root. Add the static mesh component to the default scene root, as the scene root moves, so too will the mesh.
  + We could change its relative transform, but it will still be attached to the Scene Root.
  + We could replace the default scene root and replace it with the mesh, now it becomes the root component.
  + The scene component pointer is pointing to the UStaticMeshComponent object, which overrides the root component.
* Add static mesh component in blueprint, + Add.
* The UStaticMeshComponent can be attached to other components, whereas the UStaticMesh type is a class that contains mesh information.
* Static shape meshes in starter content for instance.
  + Click and drag or use the dropdown menu.
  + Static Mesh Component has its pivot point at the bottom, a choice made by the artist.
  + Creating an Offset of the root by moving it in the BP Viewport.
* Click and drag the component to the Default Scene Root, this will override the default.
  + Cannot move roots in the BP, it will never move relative to itself.

Components in C++

* Constructing a Static Mesh Component in C++
* Class Default Object: When we create our class, item (Derived from Actor), a Class Default object is created automatically.
  + If we want to do something at the beginning of the game and be sure that all objects in the game are initialized, we must do so in begin play and not in the object’s constructor.
  + If we have a class called Bomb that explodes at the beginning of the game, we could see which actors are close to it which would cause damage to them.
* Default Subobject: To add a component, we must create default subobjects. Components are subobjects of their own actors.
* Object Type (UStaticMeshComponent etc.), internal name (different from the component variable name)
* Template functions that require a type, in this instance we can use USMC, then the name for the internal.
* Factory Functions: Functions that construct objects for us, performing internal bookkeeping.
* “That to create a new component in C++, we have to create a default sub object.”
* If something does not look right in UE after hot reloading, close the editor and recompile.
* We cannot change any location or rotation of the static mesh if it is the root component, this can only be done on an instance in the world.

Section 7 - The Pawn Class

* A pawn has additional functionality in addition to the inheritance from the actor class, can be possessed by a controller.
  + We can use key inputs to move our pawns.
  + For this example, we can create a bird, drone, or aerial like pawn which will give us a bird’s eye view of our level.
* Create a pawn parent class, we get the header file and source file. Boilerplate code included.
* - New function virtual void SetupPlayerInputComponent(class UInputComponent\* PlayerInputComponent) override; This data will allow us to move our pawn.
* Found in the same directory in the content browser, create a new blueprint class.
* Almost identical to an actor at first, adding more functionality to the drone in the next lecture.
* Don’t forget to check to marketplace for new content, just downloaded a bunch of cool assets!

Capsule creation

* Add a capsule component, this is for basic collisions.
  + Each mesh is made up of polygons, in this case, triangles. For the engine to detect when something collides with the mesh would require a lot of calculations to be done in response, which is very expensive.
  + Instead, we can use a capsule, an encompassing shape to embody the mesh and detect collision that way which requires less effort on the engine’s behalf to handle collisions.
* UCapsuleComponent, derived from UObject, not actor. UObject –> USceneComponent –> UCapsuleComponent.
  + UCapsuleComponent: Used for simple collision, can be rendered in the editor as red lines.
  + Good idea to keep variables private if we don’t plan on exposing them to other classes.
* Don’t use undefined types.
* UObject: Base class of ALL objects in UE.
* Check documentation to see if a class is stored in a specific header file, to see functions, and hierarchies of classes. Tells us what to include.
* When using includes, the class name.generated.h must always be last.
* Create a default sub object for the capsule in the constructor.
* Under shape in the BP editor, we can edit the dimensions for our capsule.
* Can define these parameters in C++ too.

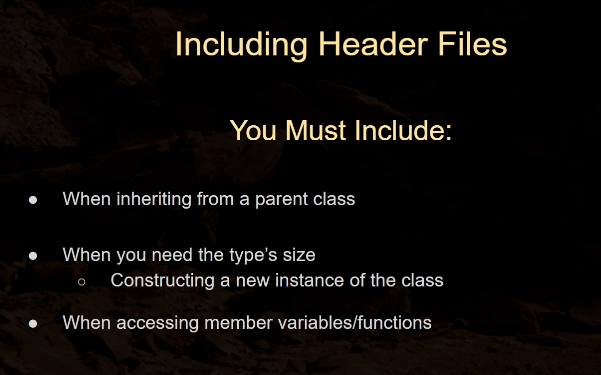
Forward Declaration

* Including Header Files: With an include statement, in this case, the capsule component class, ALL the code from the class along with other includes are going into our file, which makes it much larger.
* Better to put the includes in the C++ files, so capsule component can be defined to construct the object.
* Forward Declare: Include the class keyword in front of the class. This means we are declaring a class type without defining it.
  + Tell the compiler to go look for it somewhere else and where it needs to be included.

A screenshot of a computer

Description automatically generated

* Incomplete type is resolved by the include line at the top of the file.
* Code Bloat: When including header files where you don’t need them, classes become hugely inflated, slowing down compile time.
  + Included in CPP files makes it so that headers are put where they are needed.
* Circular Dependency: Two header files depend on each other which is not possible, this can happen if we include headers in other headers.
* Pragma once, a guard that prevents us from including the same header file twice.



* Once you forward declare, you don’t have to after it is the same class.
  + Sometimes at the top of the file near the include statements.
* Did not declare UStaticMeshComponent(), it is included by default.

Skeletal Static Mesh Component – Meshes That Have Skeletons

* UObject -> USceneComponent -> USkeletalMeshComponent (Has a USkeletalMesh variable).
  + Because it has a skeleton, it can be animated.
* SK Prefix, skeletal mesh. Opening in the skeletal editor we see its skeleton, materials, and all the bones.
* Bones are weight painted and rigged to the mesh.
  + My blender days are coming back, skeletons and weight painting is done in a DCC like Blender, not UE.
* Animation assets, much like how they are used and accessed in Blender.
  + If a mesh has the same skeleton, we can interchange meshes.
* Adding a component type USkeletalMeshComponent to the bird pawn through C++
  + Attachment for the bird mesh so that is moves with other scene components. SetupAttachment().
  + Sockets, specify the type to be attached to a socket.
* Created the mesh successfully.
* Skeletal meshes have animation settings but static ones do not, this is because static meshes cannot be animated.
  + Anim to play: Has a green line which delineates the type, animation sequence.

Enhanced Input

* Controlling Pawns
  + AController is spawned when the game starts, UE sees this and is a sort of self-insert fo the player to control the game.
  + Pawns can be possessed, the AController possesses the bird. This then allows us to configure our project ot provide input data to manipulate the pawn.
  + Pawns have a value “Auto Possess Player”. Player 0 is the first controller in the world, in a multiplayer game, the first controller is 0 and the second is 1.
  + Setting this variable to 0 means the controller will possess the bird by default.
  + UE will spawn a default sphere mesh since at first it does not see any other pawns we have configured to be possessed.
  + Multiple controllers in a game != true multiplayer. This means however many controllers exist on our machine.
  + Project Settings -> Inputs, Axis Mappings allows us to determine our inputs for a controller.
* Axis Mappings -> Move Forward -> Key or Button. ABird::MoveForward(float Value).
* Bindings, the move forward function is then called every frame of the game. Its value will vary depending on the key we are pressing.
* Move Forward will receive a numerical value while we are pressing forward, which would be 1.0.
  + If we are not pressing forward, the value will be 0.0. If it is 0, we can do nothing, if not we can move the bird forward.
  + The scale alters the value in our MoveForward function.
  + SetupPlayerInputComponent is where we do our bindings.
* & Address of operator, allows us to reference a function inside another function.

Enhanced Input Part II

* Enhanced input is a new way of using input as of 5.1, there is backwards compatibility between the old and new methods of input, but steadily, these new ones are becoming the norm.
* Using the tilde ~ in the PiE we can see input information for our character in real time.
* Input Actions (IA\_Move) are the communication between the enhanced input system and the project’s code. Indicate user input states, such as holding down a button which will trigger an action.
  + One-dimensional vs Two-dimensional input actions.
* Input Mapping Contexts: Link an input action to a specific type of input, such as linking a key to an action.
  + Enhanced Input local Player Subsystem prioritizes specific buttons to resolve collisions between multiple actions consuming the same input. Talking to a character if one is close to the player but may have a different context in another situation. “Open door” context, “Select Item” context which overrides the first context.
* Modifiers: Adjust the value of raw input coming from the user’s devices. Examples are dead zones, inputting smoothing over multiple frames. In our context, we can change the values for each input.
* Triggers: Post-modifier inputs, such as holding a button to activate a camera, then add a trigger to say, take a photo.
* Enhanced input is superior as it does not require us to go through dozens of lines of code but rather, we can manage everything from the new input system.
* Used in Blueprint, EnhancedInputAction class.
* We need to add an AddMovement component to pawns as they are not included by default.
  + FloatingPawnMovement
* Add a camera offset so we can see the character, as well as adding additional movement.
* Use a spring arm for the camera, when our camera collides with something, the spring arm shrinks to focus on the character.
* If the details panel is blank after hot reloading and creating components in C++, this is a bug that persists even in UE5, the key is to change the variable name within the C++.
* Close the editor, recompile, and delete/bring things back in.

Controller Input

* AController: Does not have a location, does have a rotation.
* Create axis mappings for our mouse as we move it.
  + AddControllerYawInput(), AddControllerPitchInput(), AddControllerRollInput()
* The controller can be a form of master rotation for the other components, it will follow it.
* Axis mapping to the mouse. Axis events. Executed every single frame.
  + Use print string to test our mapping, will return values and how fast we are moving from left to right.
* Use Negative scale to invert the looking pattern of the bird.
* Game mode base can specify the default pawns in a level.
* Setting the default pawn is beneficial, you can spawn from anywhere you choose, and the default sphere is removed.
* Use a player start actor to determine a spawn point.

Aside Before Section 8

So, we have covered a lot in the ways of implementing logic into our game, at the most basic level with vectors and debug shapes, to controlling a pawn, we are seeing how the complexity of the UE class hierarchy is getting more and more in-depth as we progress.

Before moving to the character class, I want to leave some bits of information from the UE documentation, covering some common gameplay frameworks and terminology that may come up throughout the course.

All the following comes from the official UE 5.1 Documentation:

A screenshot of a computer

Description automatically generated

Notes:

* PlayerController class is the “human player’s will” is an interesting distinction than just a controller, at their cores, controllers are actors meant to possess pawns, from there, further distinctions take place such as PlayerController and AIController, which the documentation says is a “will” that can control a pawn.
* PlayerController is typically associated with a HUD and a PlayerCameraManager, the so-called “eyeball” for a player, more on this later I assume.
* For tracking rules of a game, we can look at the GameMode, PlayerState, and GameState classes.
  + GameMode: A game’s overall concept, i.e., win/loss conditions, rules, and its DNA are parts of the game’s definition. GameMode class exists solely on the server, it does not bear much information to the client regarding data.
  + GameState: A game’s state is volatile, it includes factors such as players that are connected to a server, or where active components are in each space, achievements, missions, etc. GameState is literally just that, the “state” of the game, and this is an ever-evolving aspect of any game. GameState exists on the server and is also accessible to clients.
  + PlayerState: The state of a single player in game, either for a human or a bot that is simulating a player since this data still needs to be tracked in-game despite the latter not being a “real” player. This state could contain metrics such as a username, score, number of kills, a level, etc. All this factors into how the player exists in the current game and progress is tracked this way. PlayerStates for all players exist on all machines (unlike PlayerControllers) and can replicate freely to keep things in sync.

A picture containing text, diagram, plan, rectangle

Description automatically generated

* Connected relationship of these components in game for a possible player.
* Further gameplay framework: <https://docs.unrealengine.com/5.1/en-US/gameplay-framework-in-unreal-engine>
* A regular game is made up of a GameMode and GameState, humans that join are apart of the PlayerController class, which in turn allow the ability to possess pawns, characters, etc. which allow for physical representations in game. PlayerController also allow for players to input controls, view a HUD, and controlling the PlayerCameraManager for adjusting the camera.

UE General Terminology:

All of the C++ classes mentioned are specific to Unreal Engine.

## Project

An **Unreal Engine 5 Project** holds all the contents of your game. It contains a number of folders on your disk, such as Blueprints and Materials. You can name and organize folders inside a Project however you wish. The **Content Browser** panel inside the **Unreal Editor** shows the same directory structure found inside the Project folder on your disk.

Every project has a .uproject file associated with it. The .uproject file is how you create, open, or save a project. You can create any number of different projects and work on them in parallel.

For more information, see [Projects and Templates](https://docs.unrealengine.com/5.1/en-US/working-with-projects-and-templates-in-unreal-engine).

## Blueprint

The **Blueprint Visual Scripting** system is a complete gameplay scripting system that uses a node-based interface to create gameplay elements from within Unreal Editor. As with many common scripting languages, it is used to define object-oriented (OO) classes or objects in the engine. As you use Unreal Engine, you'll often find that objects defined using Blueprint are colloquially referred to as "Blueprints."

For more information, see [Blueprints Visual Scripting](https://docs.unrealengine.com/5.1/en-US/blueprints-visual-scripting-in-unreal-engine).

## Object

**Objects** are the most basic class in Unreal Engine - in other words, they act like building blocks and contain a lot of the essential functionality for your Assets. Almost everything in Unreal Engine inherits (or gets some functionality) from an Object.

In C++, UObject is the base class of all objects; it implements features such as garbage collections, metadata (UProperty) support for exposing variables to the Unreal Editor, and serialization for loading and saving.

For more information, see:

* [Unreal Architecture](https://docs.unrealengine.com/5.1/en-US/programming-in-the-unreal-engine-architecture)

## Class

A **Class** defines the behaviors and properties of a particular Actor or Object in Unreal Engine. Classes are hierarchical, meaning a Class inherits information from its parent Class (that is, the Class it was derived or "sub-classed" from) and passes that information to its children. Classes can be created in C++ code or in Blueprints.

For more information, see:

* [Blueprint Class](https://docs.unrealengine.com/5.1/en-US/blueprint-class-assets-in-unreal-engine)
* [Gameplay Classes](https://docs.unrealengine.com/5.1/en-US/gameplay-classes-in-unreal-engine)
* [Class Creation Basics](https://docs.unrealengine.com/5.1/en-US/class-creation-basics-in-unreal-engine)

## Actor

An **Actor** is any object that can be placed into a level, such as a Camera, static mesh, or player start location. Actors support 3D transformations such as translation, rotation, and scaling. They can be created (spawned) and destroyed through gameplay code (C++ or Blueprints).

In C++, AActor is the base class of all Actors.

For more information, see:

* [Actors](https://docs.unrealengine.com/5.1/en-US/actors-in-unreal-engine)
* [Actors and Geometry](https://docs.unrealengine.com/5.1/en-US/actors-and-geometry-in-unreal-engine)

## Casting

**Casting** is an action that takes an Actor of a specific class and tries to treat it as if it were of a different class. Casting can succeed or fail. If casting succeeds, you can then access class-specific functionality on the Actor you cast to.

For example, let's say you're making a game where you have multiple types of Volumes that can affect the player character in different ways. One of these volumes is **Fire**, which decreases player health over time. When the player overlaps with any Volume in the Level, you can **cast** that Volume to **Fire** to try to access its ""damage player health"" functionality.

* If the cast succeeds — that is, if the player is standing in the fire — the player's health starts to decrease.
* If the cast fails — that is, if the player is standing in any other kind of Volume — their health is not affected.

Casting is different from simply checking whether an Actor is of a given class, which would return a binary (yes or no) answer, but wouldn't allow you to interact with any specific functionality of that class.

## Component

A **Component** is a piece of functionality that can be added to an Actor.

When you add a Component to an Actor, the Actor can use the functionality that the Component provides. For example:

* A Spot Light Component will make your Actor emit light like a spot light.
* A Rotating Movement Component will make your Actor spin around.
* An Audio Component will give your Actor the ability to play sounds.

Components must be attached to an Actor and can't exist by themselves.

For more information, see:

* [Components](https://docs.unrealengine.com/5.1/en-US/basic-components-in-unreal-engine)
* [Components Window](https://docs.unrealengine.com/5.1/en-US/components-window-in-unreal-engine)
* [Components](https://docs.unrealengine.com/5.1/en-US/components-in-unreal-engine)

## Pawn

**Pawns** are a subclass of Actor and serve as an in-game avatar or persona (for example, the characters in a game). Pawns can be controlled by a player or by the game's AI, as non-player characters (NPCs).

When a Pawn is controlled by a human or AI player, it is considered to be Possessed. Conversely, when a Pawn is not controlled by a human or AI player, it is considered to be Unpossessed.

For more information, see:

* [Pawn](https://docs.unrealengine.com/5.1/en-US/pawn-in-unreal-engine)
* [Possessing Pawns](https://docs.unrealengine.com/5.1/en-US/possessing-pawns-in-unreal-engine)

## Character

A **Character** is a subclass of a Pawn Actor that is intended to be used as a player character. The Character subclass includes a collision setup, input bindings for bipedal movement, and additional code for player-controlled movement.

For more information, see:

* [Characters](https://docs.unrealengine.com/5.1/en-US/characters-in-unreal-engine)
* [Setting Up a Character](https://docs.unrealengine.com/5.1/en-US/setting-up-a-character-in-unreal-engine)
* [Setting Up Character Movement](https://docs.unrealengine.com/5.1/en-US/setting-up-character-movement)

## Player Controller

A **Player Controller** takes player input and translates it into interactions in the game. Every game has at least one Player Controller in it. A Player Controller often possesses a Pawn or Character as a representation of the player in a game.

The Player Controller is also the primary network interaction point for multiplayer games. During multiplayer play, the server has one instance of a Player Controller for every player in the game since it must be able to make network function calls to each player. Each client only has the Player Controller that corresponds to their player and can only use their Player Controller to communicate with the server.

The associated C++ class is PlayerController.

For more information, see [Player Controllers](https://docs.unrealengine.com/5.1/en-US/player-controllers-in-unreal-engine).

## AI Controller

Just as the Player Controller possesses a Pawn as a representation of the player in a game, an **AI Controller** possesses a Pawn to represent a non-player character (NPC) in a game. By default, Pawns and Characters will end up with a base AI Controller unless they are specifically possessed by a Player Controller or told not to create an AI Controller for themselves.

The associated C++ class is AIController.

For more information, see [AI Controllers](https://docs.unrealengine.com/5.1/en-US/ai-controllers-in-unreal-engine).

## Player State

A **Player State** is the state of a participant in the game, such as a human player or a bot that is simulating a player. Non-player AI that exists as part of the game world doesn't have a Player State.

Some examples of player information that the Player State can contain include:

* Name
* Current level
* Health
* Score
* Whether they are currently carrying the flag in a Capture the Flag game.

For multiplayer games, Player States for all players exist on all machines and can replicate data from the server to the client to keep things in sync. This is different from a Player Controller, which will only exist on the machine of the player it represents.

The associated C++ class is PlayerState.

For more information, see [Gameplay Framework Quick Reference](https://docs.unrealengine.com/5.1/en-US/gameplay-framework-quick-reference-in-unreal-engine).

## Game Mode

The **Game Mode** sets the rules of the game that is being played. These rules can include:

* How players join the game.
* Whether or not a game can be paused.
* Any game-specific behavior such as win conditions.

You can set the default Game Mode in the [Project Settings](https://docs.unrealengine.com/5.1/en-US/project-settings-in-unreal-engine) and override it for different Levels. Regardless of how you choose to implement it, you can only have one Game Mode for each Level.

In a multiplayer game, the Game Mode only exists on the server and the rules are replicated (sent) to each of the connected clients.

The associated C++ class is GameMode.

For more information, see:

* [Game Mode and Game State](https://docs.unrealengine.com/5.1/en-US/game-mode-and-game-state-in-unreal-engine).
* [Setting Up a Game Mode](https://docs.unrealengine.com/5.1/en-US/setting-up-a-game-mode-in-unreal-engine)

## Game State

A **Game State** is a container that holds information you want replicated to every client in a game. In simpler terms, it is 'The State of the Game' for everyone connected.

Some examples of what the Game State can contain include:

* Information about the game score.
* Whether a match has started or not.
* How many AI characters to spawn based on the number of players in the world.

For multiplayer games, there is one local instance of the Game State on each player's machine. Local Game State instances get their updated information from the server's instance of the Game State.

The associated C++ class is GameState.

For more information, see [Game Mode and Game State](https://docs.unrealengine.com/5.1/en-US/game-mode-and-game-state-in-unreal-engine).

## Brush

A **Brush** is an Actor that describes a 3D shape, such as a cube or a sphere. You can place brushes in a level to define level geometry (these are known as Binary Space Partition or BSP brushes). This is useful if you want to quickly block out a level, for example.

For more information, see:

* [Geometry Brush Actors](https://docs.unrealengine.com/5.1/en-US/geometry-brush-actors-in-unreal-engine)
* [Content Examples](https://docs.unrealengine.com/5.1/en-US/content-examples-sample-project-for-unreal-engine)

## Volume

**Volumes** are bounded 3D spaces that have different uses based on the effects attached to them. For example:

* **Blocking Volumes** are invisible and used to prevent Actors from passing through them.
* **Pain Causing Volumes** cause damage over time to any Actor that overlaps them.
* **Trigger Volumes** are programmed to cause events when an Actor enters or exits them.

For more information, see [Actors Reference](https://docs.unrealengine.com/5.1/en-US/unreal-engine-actors-reference).

## Level

A **Level** is a gameplay area that you define. Levels contain everything a player can see and interact with, such as geometry, Pawns, and Actors.

Unreal Engine saves each level as a separate .umap file, which is why you will sometimes see them referred to as **Maps**.

For more information, see:

* [Levels](https://docs.unrealengine.com/5.1/en-US/levels-in-unreal-engine)
* [Level Editor](https://docs.unrealengine.com/5.1/en-US/level-editor-in-unreal-engine)

## World

A **World** is a container for all the Levels that make up your game. It handles the streaming of Levels and the spawning (creation) of dynamic Actors.

For more information, see:

* [World Settings](https://docs.unrealengine.com/5.1/en-US/world-settings-in-unreal-engine)
* [Level Streaming](https://docs.unrealengine.com/5.1/en-US/level-streaming-in-unreal-engine)
* Will delve further into these links if needed, but it is good to have on the notes all in one place.

Section 8 – The Character Class

* Using Echo from Valley of the Ancient
  + To migrate, always use the content folder of the project.
  + Consult UE Samples on the marketplace to see what specific games may look like in UE.
* Character’s input: Comes with a capsule by default, as well as an arrow to show us which direction is forward for the character.
  + Capsule is the scene root for the character, a C++ variable exposed to blueprint cannot be removed from the blueprint.
  + Work with Game Mode to set Echo as the default character, recycle code form the BP\_Animal/pawn class we initialized before with our movement, this way we can control the character in much the same way. What is not inherited is the camera and spring arm by default.
* With Enhanced Input: https://youtu.be/n3n8bqu7F9Q

Moving the Camera/Spring Arm

* Use the same code as we did for our pawn class to bring over the camera with the Turning and LookUp functions to determine AddControllerYaw/PitchInput(), however for our character, we should set these values to false and check the camera pawn checkbox to allow our camera to move independently.
* These input mappings already exist from our pawn class so we did not need to do much tinkering on that end.

Moving in 4 directions

* Determine the axis mappings for left and right in the input manager, using a -1.0 scale for the left and right directions.
* Only one function needs to be created in C++ for our movement, in this case it is MoveRight(), which has the same logic as our MoveForward() function, and adding the correct movement input.
* Taking certain caveats into account with our character class, we can move around smoothly with these additions but still it is not perfect until we add the rotation, animations, and more.

Rotation Matrices

* We want to move the character in the direction our controller is facing, we need to get the forward vector of the controller via its rotation, finding the corresponding forward vector.
* We can use a Rotation Matrix; it has rows and columns with sin and cosine functions inside it.
  + We need to know the X and Y components of a particular vector moved up by 45 degrees. How do we get these components? Y = Mag (Sin (Theta), X = Mag (Cos (Theta)).
  + Multiply the matrix by a vector.

Controller Directions

* Problem with our current movement, we want to move in the direction we are facing, corresponding the controller’s rotation.
* Orient Rotation to Movement in CharMovementComp, this will turn the character towards the direction we are facing.
  + Can change the value to adjust turning speed.
* Our movement has been optimized with the usage of rotation matrices for the X and the Y, using our YawRotation, the only difference is we change the X and Y axis respectively.

Hair and Eyebrows

* Groom assets, realistic hair and eyebrows.
* A groom component is a particular class.
* Whenever you add to a build file, it is a good idea to regenerate your Visual Studio solutions, just so that everything updates correctly.

Section 9 – Animation Blueprint

* To make Echo more lively, we need to utilize an animation blueprint. This can be found in the character BP.
  + An animation blueprint requires a skeleton.
  + In the editor there is a small preview window, an event graph, and animgraph, which handles poses. The event graph is the logical portion, whereas the animation graph is for poses and the ways bones are manipulated.
  + Asset browser shows you the animations rigged to the skeleton we chose.
* Environment note: Floating objects?
* Soft versions of the variable, “lazily loaded”.
* Object reference: Like a c++ reference to an object.
* Class reference: Holds a type, references a class.
* TryGetPawnOwner, very common and will allow us to get the value of a variable.
* Event Blueprint initialize animation, begin play.
* “Flow of Execution”
* Promoting variables.
* Velocity is a vector.
* If we want to switch from an idle animation to a moving animation, how is that done?
  + Create two different states that we can move in between, known as “animation states”.
  + Create a state machine:
  + Arrwos between nodes in the state machine are transition rules, which outline conditions for us to go between animations and elements.
  + Our state determines our return on the state machine, returning a pose.
  + Go into the “idling state” and you can put an animation into that pose.
  + Blending between states in the state machine, 100% in a state means that animations is solely taking place.
  + Result was visible but ignored, means there was no data in the state.
  + Booleans in transition rules, once the condition is true or false, then that state is satisfied.

Create a C++ class to be the animation BP parent

* AnimInstance is what all animation blueprints are based off of.
* Anim blueprint is updating all the time, as soon as we make our C++ as a parent, it will be updating all the time as well.
  + Best to compile, crashes can be very easy. Close the editor before we make changes to the animation instance.
* Defining the same begin play functions in C++.
* To expose to the event graph, we use BlueprintReadOnly UPROPERTY.
* UE has its own cast function.
* Update ground speed every frame.
* Kismet Math Library, we can use functions on it. It is a static library that contains static functions. Has a ton of math functions for us to utilize.
* If files get corrupted, we can regenerate VS project files (bullshit with deleting intermediate saved etc.)
* Change the anim instance to our C++ class.
* Show inherited variables in the My Blueprint section to see our C++ variables in BPs.

Adding a jump to Echo’s move set

* Action mappings have a name, so we can call this one jump for example with a key binding, just like axis mappings.
* Action mappings are **not executed** every frame but are a one-off. We can bind a function to our action mapping, the callback function will be called one time, which executes.
  + Action mapping is more efficient, if we do not need something to happen every single frame, we can assign action mappings instead of axis.
* From the input settings page directly: “Action and Axis mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a layer of indirection between the input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range.”
* Can be accessed directly from blueprints in the vent graph.
* The jump function is built directly into the character.
* Bind axis and action in C++.
  + IE\_Pressed and released, enum type that determines if we run the action when the key is pressed or released.
* No animations on jump yet but the built-in function works as it should. We need to add jump states.
* Use cached poses to transfer state machines to others, storing the poses and the data along with them.
* When creating new states with a solitary animation, just drag it into the graph raw.
* NOT Boolean operator in Blueprints, negates.
* Automatic rule based on sequence player in state, returns to the default animation, in our case this is our ground locomotion.

Inverse Kinematics

* On slopes, our character will appear to hover. By using Inverse Kinematics, we can fix this problem.
  + A method of solving equations so that we can move a particular bone of the skeleton, emerging from robotics such as controlling joints and end points.
* To make this look more natural, we would make the legs bend using IK, pushing the character down the slope to make it look more realistic.
* IK handles the equations required to make this happen, such as how far to make the legs come up and where to adjust them.
* Sphere tracing: Starting at the end point with a sphere, can detect hits so we can tell how far we trace from the end point to the ground.
  + We perform two sphere traces at once, one on each leg.
  + Push the character down by a certain number of units via the sphere tracing.
  + ZOffset\_L and ZOffset\_R, which one is lowest? Once the lower offset is found, we can move the bone down by that amount.
  + Interpolation: We smoothly move towards a target value so that we see more natural movement, especially important for IK on a character skeleton if we are going for realistic proportions.
* UE has IK built in.
* End Effector: The bone we are moving, and the bones attached to it will move simultaneously.
  + Design a control rig.
  + Specify a skeleton for a control rig. Rig hierarchy -> Import hierarchy
  + Use IK bones found on the Echo skeleton, these are not rendered as meshes but rather we can just use them for inverse kinematics.
  + IF the character does not have IK bones, use virtual bones.
* Once we import the skeletal mesh, we can start by creating our sphere trace.
* We can move the IK bone and the mesh will not move along with it, once we do more calculations, we can move the mesh properly.
* RECAP:
  + Made a Boolean, if true, we do a sphere trace from above the foot to below it, only concerned with the Z value on both the right and left foot.
  + Take the z offset variables and interpolate it, move it smoothly towards the target from our sphere trace.
  + See which is the lower value, then move the pelvis down.
  + Transform the IK bones, adding the z offset to the left, right, and pelvis.
  + Full body IK, bending the other bones in the correct way.

Chapter 10 - Collision

Overlap Events

* ActorBeginOverlap: Built into all actors by default, we can use it to trigger something in response to an overlap from another actor. Overlapping with components, if they are configured to respond to the events, something will happen.
* If we set an object type to overlap, say a pawn overlapping with a static mesh component, we can trigger events to take place.
  + I imagine damage calculations are done in this exact same way, once the collisions meet their conditions, something will happen.
* Overlapping not with the actor itself but collision components, i.e., using a sphere component that is far larger than the actual mesh itself.
* Individual overlap events for components themselves, such as On Component Begin Overlap.

Delegates

* Design Patterns: Solutions to commonly found problems in software development. Reused in future programs.
* Observer Pattern: Observers are pointers to other objects within a program, each of which has a callback function. Observers want callback functions to be called in response to something happening in the program. That something is an event, and this design pattern is for observers to call the callbacks in response to an event. Knows when the event happens, then goes to the callbacks.
* A delegate is a special class, containing a delegate list which has observers inside it. The delegate loops through the list and calls the callback functions on the objectsA screenshot of a computer

  Description automatically generated
* Must have the same input parameter, once these functions are bonded to the SummonMinions delegate, the callbacks are created. A value can be passed in to satisfy the certain parameters found in each function.
* Overlap Events use delegates as well,

On Component Begin Overlap – Delegates Expanded

* Ctrl – k – o, open the cpp file derived from the header file.
* When you are trying to create an object, forward declare, then use pointer? That seems to be the pattern. Then in the CPP file, you create the object in the constructor.
* Making collision object start and end events, simple with the meshes.
* Next up is creating collision events for the sword or items that we choose to implement.

Section 11 – The Weapon Class

* We have had classes derived from each other throughout the course, such as AActor, AItem, and now, AWeapon.
  + Item has certain properties, such as using sinusoidal movement, overlapping, and having a mesh. The weapon class will take all of these but will include new capabilities such as attacking and equipping.
* Create C++ class derived from the original.
* Make overrides without the UFUNCTION().
* Add the override at the end of the function declaration.

All About Sockets

* If we want to attach an actor to another one, we need to use sockets.
  + Sockets can be connected to skeletal meshes.
  + Right click on any point of the skeleton and click “Add Socket”.
* Because the socket is relative to the hand bone, it will move along with it.
* Sockets for characters at least are surprisingly simple, though I may not know the half of it just yet. This could be used for a variety of actions though.

Downloading Animation Assets

* Added the weapon and preview mesh to the skeleton.
* Need animations for holding and attacking with the weapons.
* Mixamo, back again…
* No smoothing group errors, not a problem as this will always happen when importing, according to Stephen.
* SKM prefix = Skeletal Mesh, which is *not* the same as a skeleton asset.
* Mixamo animations are made directly for the mixamo skeletons, so importing this one is necessary.

IK Rigging

* Retarget root, a frame of reference for all the other bones in the IK rig.
* Each body part has a chain of bones.
* Making chains, I have no idea what this is for yet.
* I figured it out.

IK Retargeting

* Retargeting mixamo animations over to Echo

Attaching the Weapon Mesh

* Attach the root component to our hand socket, via blueprints or C++.
* Attach Component to Component function in BP.
* When we attach, we need to determine rules for it. We want to snap to the target and not keep relativity.

IF you receive the error, Blueprint is derived from a corrupted class, delete it and carry over everything to a new one, don’t forget to compile after you do this. Go into the sln and explorer files and make sure the original is deleted so the build succeeds.

LINKS

* UE Coding Standard: <https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/>
* Configuration for VSC for UE: <https://docs.unrealengine.com/5.0/en-US/setting-up-visual-studio-code-for-unreal-engine/>
* UE5 Documentation: <https://docs.unrealengine.com/5.0/en-US/>