Day 2 - 1/01/2023

rotator system - multiple type of coordinate system

LHS or RHS

how local and world axis differ from each other

yaw - rotation along z axis

roll - rotation along x axis

pitch - rotation along y axis

now Let's start the main I started this game dev c++ Game development so first we need to instal the IDE for c++ there are some paid software by Jet Brains for unreal engine called Ride and some program by visual Assists x called Tomato whole tomato

C++ Refresher

The majority of these concepts involve classes and inheritance. These are the main principles of object-oriented programming, and these are what give object-oriented programming its strength.

Imagine that you have a C++ class.

Now classes have their own variables and also their own functions. And this class can be a parent to some child class, which derives from that parent class. So, each child class will inherit variables and functions from the parent class, as well as possibly having its own child variables and child functions.
Now, the parent class has the option to mark some of its functions as virtual. This means that child classes can override those functions, creating their own child versions of those functions.
Now the child can choose to call its own custom version of the function, its override, but it also has the capability of calling the parent version of the function So it still has access to that parent version.
Now, a parent class can have a child class, and a child class can have its own child class, and this can go on indefinitely.

As long as a parent has a child which has another child and so on. You have what's known as an inheritance hierarchy or also known as an inheritance chain. Now a class can have many children. In fact, there can be many child classes that derive from the same parent. This happens all the time, so an inheritance hierarchy can get quite complex very quickly.

A child class can also inherit from multiple parents. This is known as multiple inheritance. So, in this case, each parent will have its own unique variables and functions and the child class will inherit variables and functions from both parents. Now, when you have a given inheritance hierarchy, you might create a variable of type pointer to parent. A pointer is a variable designed to hold an object's address. So, a variable of type pointer to parent is capable of

storing the address of an object derived from the parent class.

When a pointer stores the address of a given object, we say that it points to that object. A variable of type pointer to parent can point to an object of the parent class, but as you may know, a pointer to parent variable can also point to an object of a child class, provided that that child class inherits from the parent class. So, our variable of type pointer to parent is capable of storing the address of an object of any type in this inheritance hierarchy.

As long as it's derived from parent or parent, at least exists somewhere up the inheritance chain.

Now, the parent class might have a function, and thanks to inheritance, each child class lower in the hierarchy will inherit that function. Now, if this function is marked as virtual in the parent class, that each child class can have its own unique override

of that function.

Now, if you have a variable of type pointer to parent and it's pointing to an object of one of the child classes, you may access the pointer and call this function. Now, the child override version of the function has the same function name as the original, the one that's declared in the parent class. So which version do we get called when we call the function? The answer depends on the object that our pointer is pointing to. Since the pointer is pointing to a child in this case, then when we call the function, we will get the child version called. So even though the pointer itself is a pointer to parent variable, it's the object pointed to that determines which version of the virtual function that gets called. This phenomenon is known as polymorphism.

Now consider this inheritance hierarchy. Let's say we have a pointer of type pointer to child one and we call

this variable first child. And let's say that we have this pointer pointing to a child three object.
This is legal in C++. We can have a pointer to a parent class. In this case, Child one is one of the parents of child three and the pointer points to an object of the child three type.
Now let's say that child three has its own function, which is not defined in the child one class. And it's not an override. It's a function that belongs solely to the child.
Three class. We know we have an object of type child three but we're storing its address in a child one pointer. We can't call this function because the compiler isn't going to know that our object has this function. It sees that our pointer type is child one and child one doesn't have this function.
So how do we call this child three function? Well, what we can do is create a new variable of type

pointer to child three and we can make use of a special type of function called a cast.
A cast will take our variable of type pointer to child one and convert the type to pointer to child three. This cast happens at runtime, so it requires an ability known as runtime type checking, which means the object itself must be inspected for its actual type. If we're trying to cast this pointer variable to a child three pointer, the object itself must really be a child. Three objects. In our case it is. Now. If it is, the cast will succeed and the cast function will return a value of type pointer to child three which we can then store in our new pointer to child three variable. If the object itself is not actually a child three object, the cast will fail and return null.
Once we successfully cast from child one to child three, we now have a variable of type pointer to child three which we can then use to call the function on

the child three class. Since we casted our pointer from child one down to a child three pointer, this type of cast is called a down cast.

In C++, you've learned about static casts and dynamic casts. Static casts are performed at compile time and can be used when you know the types you're casting to and from at compile time, like casting an integer to a float, for example. But in this case, at runtime, we can't really know what our child one points to, whether it's an object of type child one child two or child three that can depend on what's happening in the game as these are dynamically instantiated.

So that can depend on what's happening in the game. So that's why runtime type checking is necessary. And this type of cast is known as a dynamic cast, since it can't be performed at compile time.

Whenever we do a dynamic cast, it's important to

check the pointer afterwards to make sure that it's not a null pointer as this can be the case if the cast fails. Unreal Engine has its own inheritance hierarchy. Near the top of the hierarchy is a class called US Object. We typically refer to instances of this type as objects derived from the object class is the actor class. We call these actors. Further down the hierarchy, the class a pawn derives from an actor, and further down than that, the class a character derives from upon everything derived from you object, which is not an actor, has you prefixed on its class name. That way we can take a look at the class name and know that it's not an actor but a you object. Everything derived from actor has its name prefixed with an A, so we can take a look at the – and character classes, for example, and know that they're inherited from the actor class. The object class is very basic.

It can store data, but it doesn't even have the ability to be placed in the world by itself. That's how basic it is. The actor class inherits everything from the object class, but it has some significant upgrades. For one, it actually can be placed in the level. So, everything you see in the game level is derived at least from the actor class or below. Actors can have a visual representation such as a mesh, and they have many more capabilities that we're going to learn about in this course.

Pawns are derived from actor, but they have even more capabilities. The most significant perhaps, is that they can be possessed by a controller. Now, a controller is a type of actor designed for controlling pawns. When you press the key and your character moves forward, that's thanks to the fact that the. Troller is receiving your input, processing it, and

turning that information into the action of moving your character forward. So, since pawns can be possessed, this is a good class to choose for things that need to move around based on some form of input, whether it's from a player pressing keys and moving the mouse or input calculated by the means of artificial intelligence derived from --. We have the character class since it inherits everything from the --class. The character class to has the capability of being possessed by a controller. It also has more functionality more suitable for bipedal creatures like humans. It has a character movement component which performs all sorts of calculations related to moving characters. So, characters are just special type of pawns that have more character specific functionality as well. In general, if you need a very basic type of creature that simply needs to take input and move around, a -- is enough, but if you want more

character specific functionality, you'll choose a character. Throughout this course, we'll be demonstrating that extra functionality that characters have once we create a character of our own.

Now the object actor pawn character hierarchy is actually more complex than it looks here. There are many different classes derived from each of these. These are just several examples, but there are many more. Another important aspect of object-oriented programming is the difference between is a and has a relationship. Imagine you have a child class and that child derives from some parent class. It's said that the child is a parent because the child inherits from that parent class. And this is why the child contains all the functions and variables that the parent contains through inheritance.

The parent, however, is not a child. So, this

relationship does not work both ways.
Now a child class can have its own child. We can call this grandchild. In this case, we can say that the parent is not a child and the parent is also not a grandchild. And we can say that the child is a parent, but the child is not a grandchild. We can say, however, that the grandchild is a child and is also a parent. It inherits functions and variables from both the child and parent classes.
Likewise, in the Unreal Engine class hierarchy, we can say that the new object class being the parent of the actor class, is not an actor, and the new object is also not a pawn. But the actor.
Being a child of the object class is an object, but the actor is not a pawn. And the pawn being the child of actor is both an actor and an object. This explains is a relationship. Now let's discuss has a relationship.
Imagine you have a class and a class can have its own

variables. Some of those variables might be custom data, types of various classes or structs. So, inside your class you might have a member variable that is itself another class We can call the first class the outer class, and the variable contained inside of that class can be called the inner class. Now, the inner class can have its own variables, some of which themselves have data, types of other classes and structs. So inside of an inner class you can have an inner inner class and unreal engine classes are nested within one another quite frequently. At the outer level of a game project, you have a class called package inside of the package class as nested.
Another class in this class is called The World Inside of the World Class as nested yet another class and this is known as the level. Now objects are placed in our level and these are classes that are derived from actor or lower in the inheritance hierarchy.

So, a level contains actors. Now, actors typically have their own variables, some of which are components. Components are a type of subclass designed to be owned by actors, and they provide additional features that the actor can use. So, an actor can have one or more components.

So, it said that a package has a world and a world has a level, A level has actors and actors can have components. It said that an actor component is a sub object of its owning actor, and the actor is a sub object of the level. The level is a sub object of the world, and the world is a sub object of the package. Now there are some objects that are not nested inside the package like all of these classes, and these tend to be assets such as meshes, textures, sound files and so on. So, these are nested inside of the package itself and we don't refer to these as sub objects. So