CS330 Project 02 Guide

Spring 2019

By: Gurman Gill, Sonoma State University

Contents

Acknowledgement

This project has been created in collaboration with Sanjay Madhav, University of Southern California.

Project 02: TopDownShmup

To get more in-depth experience with Unreal 4, we're going to make a top-down shoot 'em up. You'll have a character that runs around and shoots at enemies. Please make sure you have completed the Unreal C++ Programming tutorial video series (Lab 06, 07) – otherwise, it will be very difficult to follow these instructions.

This project is divided into 4 parts. The difficulty level increases linearly: the document has a lot of hand-holding in the earlier parts, as it progresses, I take off the training wheels. So don't be tricked into thinking you can just procrastinate until right before it's due – there is a decent amount of work that is involved, so you will want to work on it regularly.

In any event, let's get started! Download the TopDownShmup.zip file from Canvas and extract the files. Inside this folder, open the TopDownShmup folder. Then, follow these instructions (depending on your platform):

Mac

1. Right click on TopDownShmup.uproject and select "Services>Generate Xcode Project"
2. Open the resulting TopDownShmup.xcodeproj
3. In Xcode, in the top toolbar and to the right of the Stop button click on "TopDownShmup->My Mac"
4. Edit scheme to change the build configuration to "DebugGame Editor".
5. Click the Play button. This will build the project and open up the editor.

Windows (Disclaimer: I haven't tested these)

1. Right click on TopDownShmup.uproject and select "Generate Visual Studio Project files"
2. Open the resulting TopDownShmup.sln file
3. Select the "DebugGame Editor" configuration
4. Run the project in Visual Studio, which will build and then open up the editor

Once this is setup, you should be able to play the game in the editor. You'll notice the character moves around by clicking the mouse. This is essentially the "Code Top-Down" template, except there is a fancier model taken from one of the sample games. Now we are ready to code.

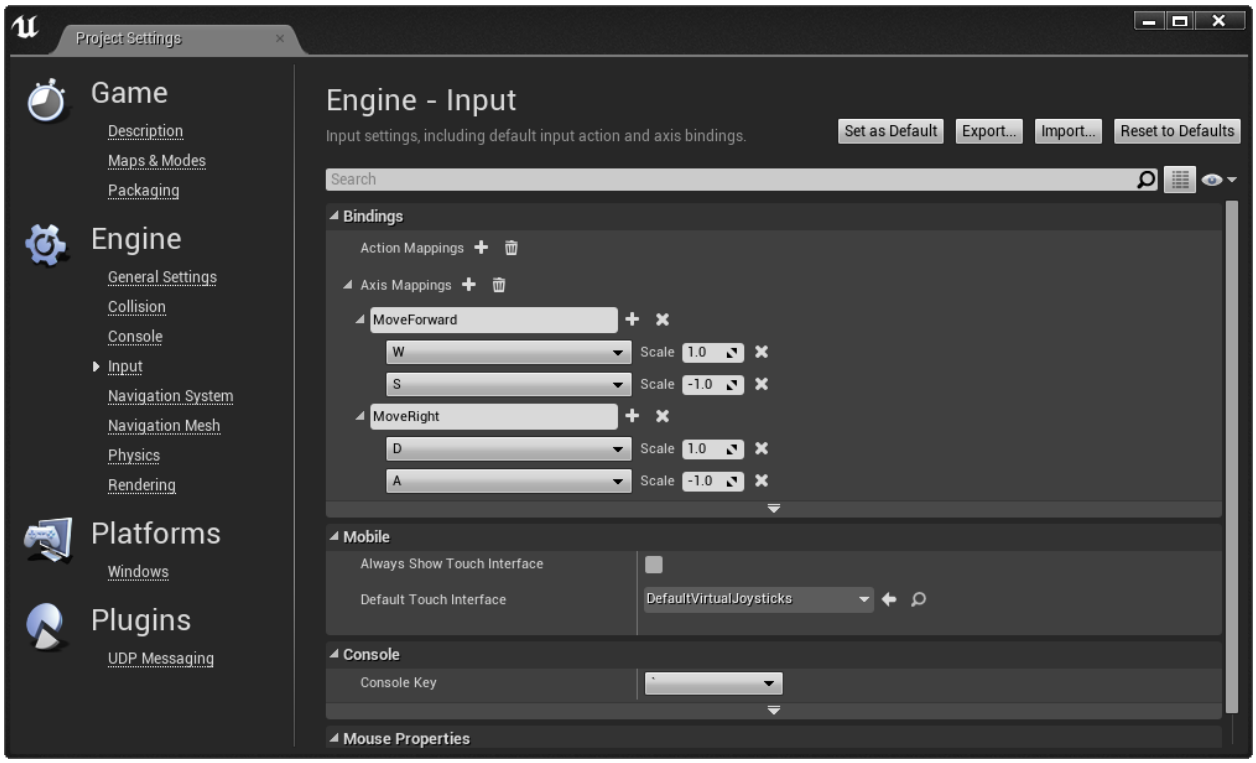
Part 1: Movement and Facing

The first thing we're going to do is change the way the character moves. Click-to-move works well for an RTS (Real Time Strategy), but it's not great for a SHMUP (SHoot'eM UP) game. We'll change it so the character moves with WASD and the mouse is used to turn and aim the character.

The player controls are all handled in TopDownShmupPlayerController.h/cpp. Take a second to look through the code in this file, to get a sense what every function is doing.

Now, let's disable the mouse movement. The `PlayerTick` function is called for a local player on every frame. If you comment out the `MoveToMouseCursor` call, it'll no longer move. Test it out and verify that you have succeeded in breaking player movement. Good job!

In the Unreal Editor, if you open up Edit>Project Settings under Engine/Input you'll see that there are already two axis mappings for `MoveForward` and `MoveRight`, which are mapped to WASD. It looks something like this:



So since the WASD mappings already exist, we might as well use them.

Back in code, declare two new void functions in the Player Controller header. Call these functions `MoveForward` and `MoveRight`. They both take in one parameter, a `float` called `Value`.

Then in the Player Controller cpp file, you need to add axis bindings to `SetupInputComponent`.

It'll look like this for `MoveForward`:

```
InputComponent->BindAxis("MoveForward", this, &ATopDownShmupPlayerController::MoveForward);
```

Do the same thing for `MoveRight`.

Now you need to actually implement the `MoveForward`/`MoveRight` member functions.

The `MoveForward` function will look like this:

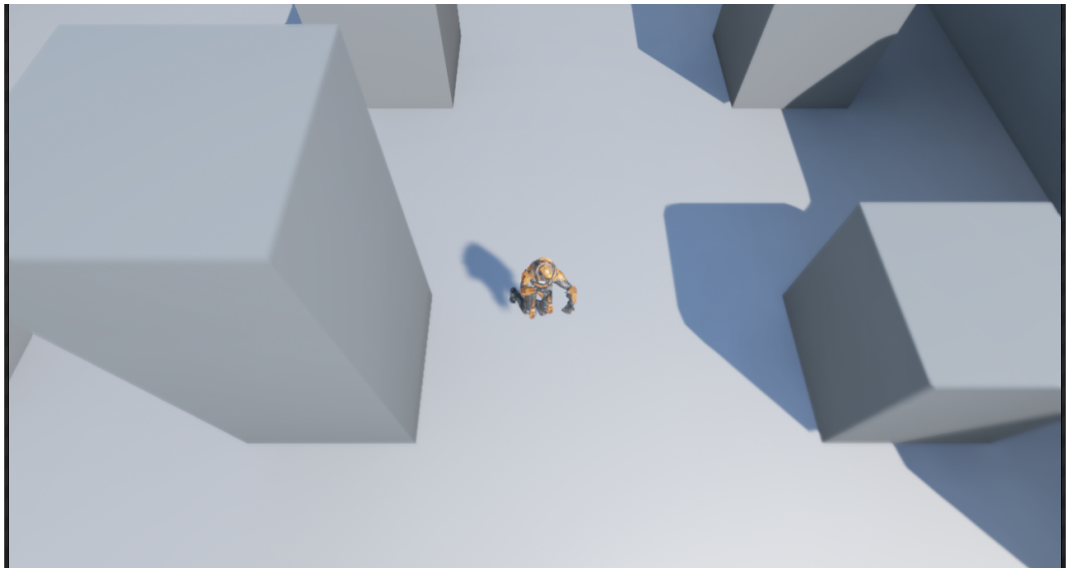
```
void ATopDownShmupPlayerController::MoveForward(float Value)
{
    if (Value != 0.0f)
    {
        APawn* const Pawn = GetPawn();
        if (Pawn)
        {
                Pawn->AddMovementInput(FVector(1.0f, 0.0f, 0.0f), Value);
        }
    }
}
```

What this does is if the `Value` is non-zero, it tells the player to move along the global forward vector, which in our case is along the x-axis.

Your `MoveRight` member function should basically be the same, except you want to move along the y-axis.

Now test out your WASD movement. In the editor, go to the dropdown to the right of "Play" and select "New Editor Window." This way, when you click Play from now on, it'll open up in a separate window that automatically has focus. Otherwise, you have to manually click on the window for it to begin accepting keyboard input.

You should now be able to move around your character using WASD. Notice how the character also automatically turns to face the direction he's travelling in.



Now let's implement mouse look. Create a new function in the controller called `UpdateMouseLook`. Inside this function, the logic should be something like this:

1. Get the `Pawn` pointer
2. If there's a `Pawn` pointer, call `GetHitResultUnderCursor`, just like how it was called in the provided `MoveToMouseCursor` function.
3. If `Hit.bBlockingHit` is true, then:
    a. Construct a new `FVector` that goes **FROM** `Pawn->GetActorLocation()` **TO** `Hit.ImpactPoint`. Remember that this means vector subtraction.
    b. Set the z-component of this vector to `0.0f`. This is because we only care about rotation on the xy-plane.
    c. Normalize this vector (`FVector` has a `Normalize` member function – a reference with all the `FVector` member functions is here)
    d. Convert this vector to an `FRotator` using the `Rotation` member function
    e. Call `Pawn->SetActorRotation` with this new rotator passed into it

Your `UpdateMouseLook` function should be called by `PlayerTick`. Once this is implemented, you should have the character turning to face the mouse. The animations admittedly don't work perfectly, but it's good enough for our purposes.

Part 2: Adding a Weapon

Right now, our character is in a pose where it seems like he should be holding a weapon, but he's holding nothing. So let's make a Weapon class. Go to File>Add Code to Project and create a class called Weapon that inherits from Actor.

First, let's add stuff to Weapon.h. For now, we want to have only one property: a `USkeletalMeshComponent*`. Call this `WeaponMesh`. Set the `UPROPERTY` to `VisibleDefaultsOnly`, `BlueprintReadOnly`, and `Category=Weapon`. If you don't remember how to do this, I'd recommend opening up the project made for second video of Lab06 and looking at Pickup.h/cpp.

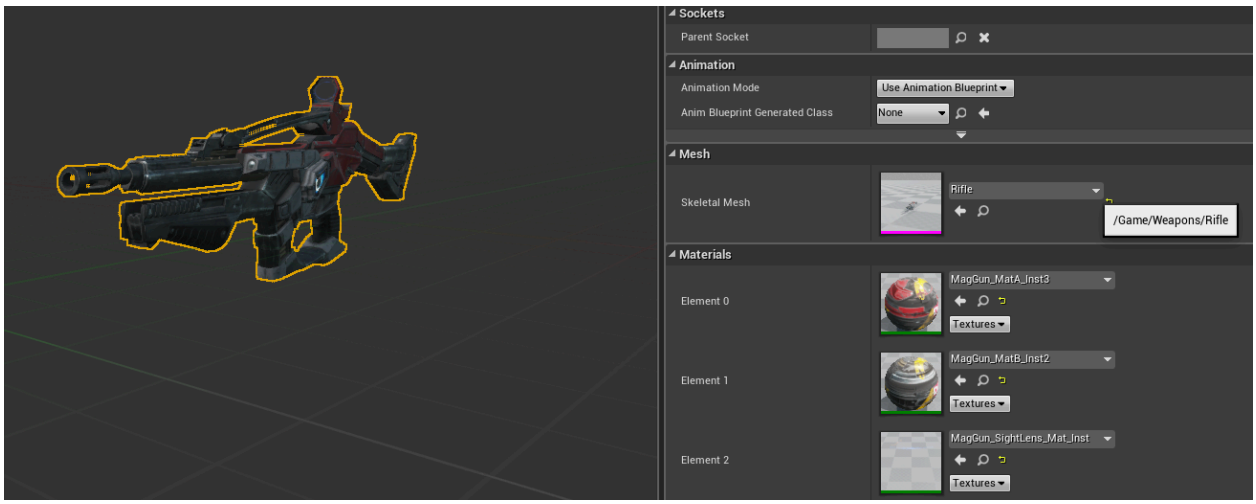
Next, create two (public) `virtual void` functions called `OnStartFire` and `OnStopFire`. These will be called when the weapon starts and stops firing, respectively. Don't worry about giving these `UFUNCTION` attributes – these are going to be C++ only functions.

Then in Weapon.cpp, you need to initialize the `WeaponMesh` (using `CreateDefaultSubobject`). Set the `RootComponent` to `WeaponMesh`. Then add empty implementations for `OnStartFire`/`OnStopFire` – we'll add the code for these functions later.

Hopefully, everything builds. Go back to the editor and now create another C++ class called `AssaultWeapon`, which should inherit from your `Weapon` class.

In the code for `AssaultWeapon`, you'll want to override the `OntartFire`/`OnStopFire` functions from the parent `Weapon` class. Add implementations for these functions that call the `Super` functions, but don't do anything else for now.

Run the editor once again, and if everything worked you should be able to create a Blueprint from the `AssaultWeapon`, I called it `BP_AssaultWeapon`. Set the `WeaponMesh` to the "Rifle" mesh. It should look something like this:



Attaching the Weapon

Now that we have a weapon, we need to actually spawn it and put it into the player's hands. In TopDownShmupCharacter.h, add a `TSubclassOf<AWeapon>` called `WeaponClass` with a `UPROPERTY` of `EditAnywhere` and `Category=Weapon`. This is what we'll use to define what type of weapon we want to attach to the character. Next, you need to make a private pointer to a `Weapon` called `MyWeapon` that we'll use to actually keep track of the weapon instance. Don't forget you need to include Weapon.h (before the generated.h) or forward-declare `AWeapon`.

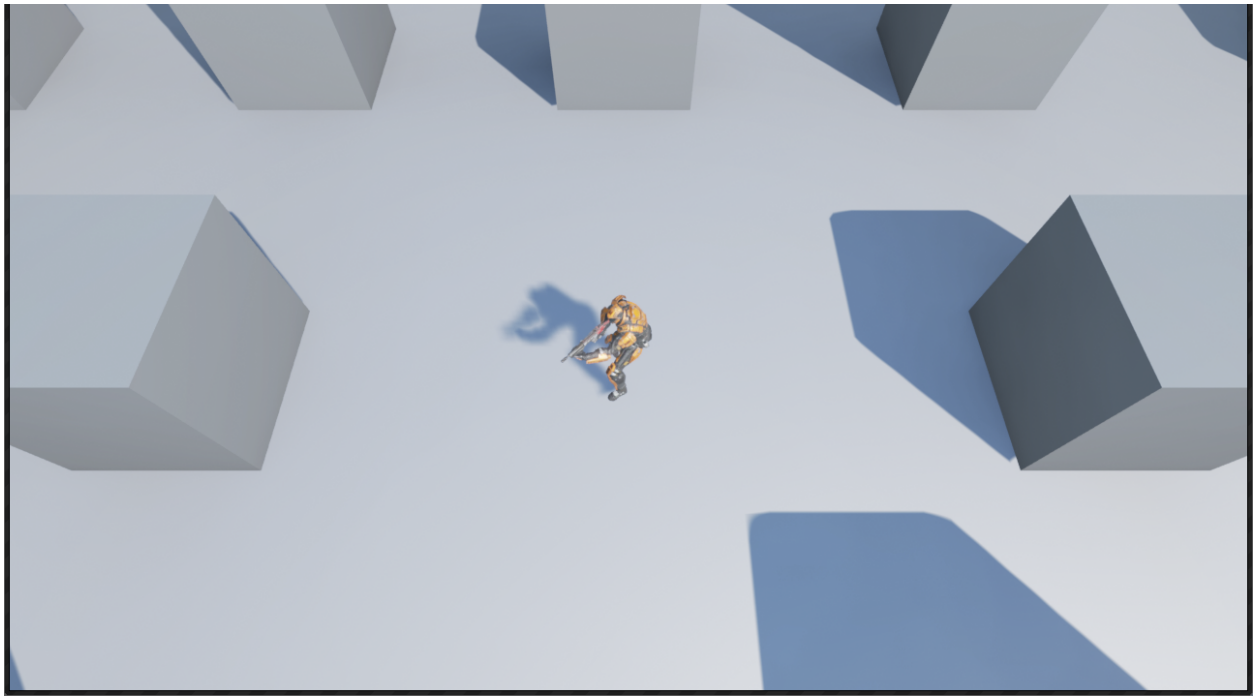
Next, we need to override the `BeginPlay` function, which is the function called once the World has spawned and it's time to start the game. This is where we want to create the weapon.

The code for spawning the weapon is very similar to spawning the pickup in the tutorial. However, there is one addition that must be made in order to attach the weapon to the character's skeleton. This ends up looking like:

```
void ATopDownShmupCharacter::BeginPlay()
{
    // Call base class BeginPlay
    Super::BeginPlay();

    // Spawn the weapon, if one was specified
    if (WeaponClass)
    {
        UWorld* World = GetWorld();
        if (World)
        {
            FActorSpawnParameters SpawnParams;
            SpawnParams.Owner = this;
            SpawnParams.Instigator = Instigator;

            // Need to set rotation like this because otherwise gun points down
            // NOTE: This should probably be a blueprint parameter
            FRotator Rotation(0.0f, 0.0f, -90.0f);

            // Spawn the Weapon
            MyWeapon = World->SpawnActor<AWeapon>(WeaponClass, FVector::ZeroVector,
                Rotation, SpawnParams);
            if (MyWeapon)
            {
                // This is attached to "WeaponPoint" which is defined in the skeleton
                // NOTE: This should probably be a blueprint parameter
                MyWeapon->WeaponMesh->AttachTo(GetMesh(), TEXT("WeaponPoint"));
            }
        }
    }
}
```

You'll probably notice that this code is not the most robust. I've hard-coded both the rotation and the attachment point, which clearly is not a good coding practice to follow.

In any event, open up the editor and set the "Weapon Class" in the `TopDownCharacter` blueprint to `BP_AssaultWeapon`. Your player should then run around with a weapon.



Firing Effects

A weapon that doesn't fire isn't very useful. But before we add in firing, let's remove everything related to "SetDestination" that's in the player controller, since we won't be using it in our game.

Once you've verified the game still builds and runs, open up the input configuration in the editor. Rename the SetDestination action mapping to Fire (since it's already mapped to the left mouse button).

Now in the player controller create two functions – `OnStartFire` and `OnStopFire`, which should be mapped to `IE_Pressed` and `IE_Released` for "Fire". These functions should get the pawn and cast it to our character class type. Then they will call `OnStartFire` and `OnStopFire` functions on the character (which you'll have to declare). Finally, these functions in the character class should call `OnStartFire`/`OnStopFire` on the weapon, if there is one. It's an annoying

sequence of function calls, but it is proper encapsulation – we don't want to allow the player's weapon to just be messed with from anywhere.

Once you've verified that this runs (you could even use breakpoints in the debugger to verify your AssaultWeapon's `OnStartFire` function is getting called), it's time to actually make something happen when you fire.

In Weapon.h, declare two member variables of type `USoundCue*` and call them `FireLoopSound` and `FireFinishSound`. These should be `UPROPERTY EditDefaultsOnly` and `Category=Sound`. We will use these to define the sound cues for both the firing looping sound and the sound to play when we stop firing.

Next, create a protected member variable of type `UAudioComponent*` called `FireAC`. This should be `UPROPERTY Transient`. An audio component is created when you start playing a sound cue – in our case, we'll need to grab the audio component of the fire loop so we can later stop it.

Finally, create a protected member function that returns `UAudioComponent*` and takes in a `USoundCue*` called `PlayWeaponSound`, which I "liberated" from the `ShooterGame` sample:

```
UAudioComponent* AWeapon::PlayWeaponSound(USoundCue* Sound)
{
    UAudioComponent* AC = NULL;
    if (Sound)
    {
        AC = UGameplayStatics::PlaySoundAttached(Sound, RootComponent);
    }

    return AC;
}
```

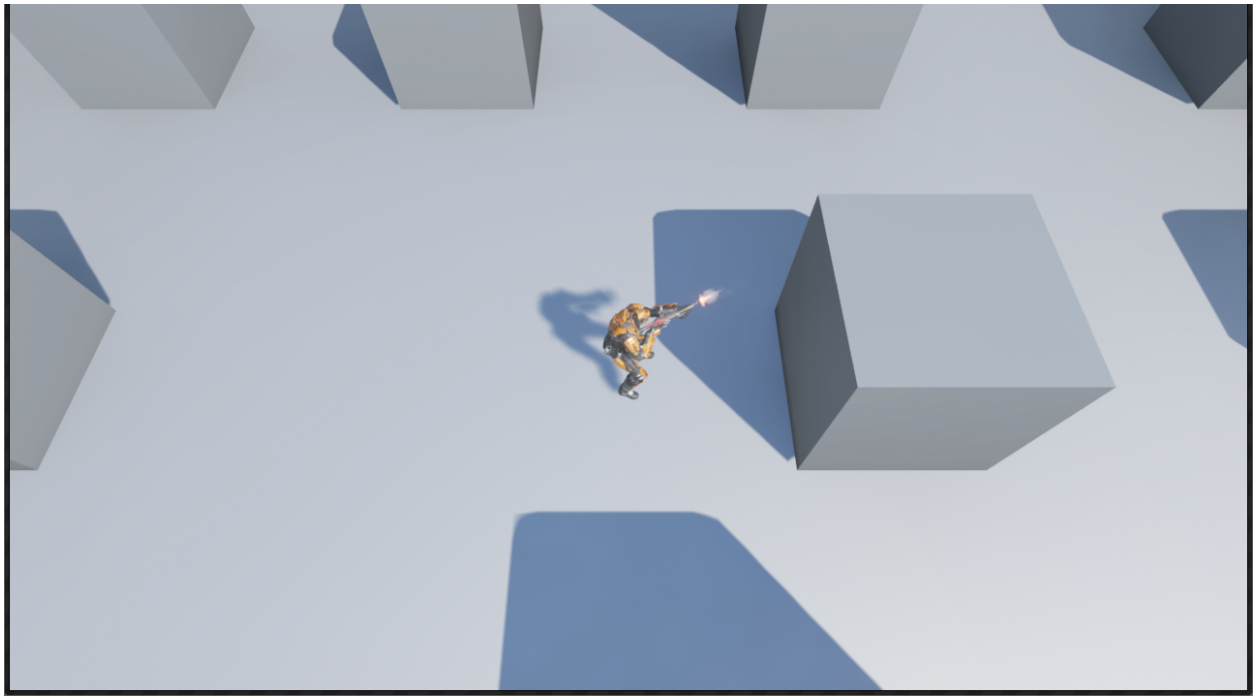
Then make `OnStartFire` start playing the loop sound and save off the AC. Then make `OnStopFire` stop the looping sound AC and play the `FireFinishSound` (you don't need to save the AC for this sound, because it doesn't loop).

If everything builds (You may need to add Engine.h in your Weapon header file), you should then open up `BP_AssaultWeapon` in the editor and set "Fire Loop Sound" to `AssaultRifle_ShotLoop_Cue` and "Fire Finish Sound" to `AssaultRifle_ShotEnd_Cue`. If you play the game now, you should hear firing sounds (warning: it might be a little loud).

Now that it sounds like firing, let's add a muzzle flash effect. This effect is looping, as well. So you create a `UParticleSystem*` member variable called `MuzzleFX` with `EditDefaultsOnly` and `Category=Effects`. Then also make a `Transient UParticleSystemComponent*` just like you did for the `FireAC`. You can spawn the particle system with

`UGameplayStatics::SpawnEmitterAttached`. For the attach point, pass in `TEXT("MuzzleFlashSocket")`. Then stop the particle system with `DeactivateSystem` when you stop shooting. You'll need to include ParticleDefinitions.h in the .cpp file, as well.

Once the code is done, update `BP_AssaultRifle` so the `MuzzleFX` is set to `P_AssaultRifle_MF`. If this all was copacetic, you'll have a muzzle flash.



Of course, the weapon doesn't actually fire anything yet, but at least it looks and sounds cool!

Actually Firing

Generally speaking, there are two ways to implement firing weapons in a game. One way is to spawn an actual projectile object and then simulate the projectile as it travels through the world. This is preferred for slower projectiles, like `MuzzleFlashSocket` s, or faster projectiles that just travel very far. Another way is to instantly fire a ray from the gun, and see if the ray intersects with anything. If it does, then you hit the object.

Since our current weapon is an assault rifle, we'll go with the second approach. We'll want this ray cast to happen every *x* seconds, where *x* is the rate of fire. If the ray intersects with an object, we'll do damage to it (if it takes damage) and spawn a bullet hit particle effect. However, since

this is only specific to an assault rifle type weapon, we want to do this in `AssaultWeapon` and not the base `Weapon` class.

But before we make changes to `AssaultWeapon`, there's one other thing we need to add in the base `Weapon` class – a pointer to the `APawn` that owns the weapon (call it `MyPawn`). Then in `ATopDownShmupCharacter::BeginPlay`, set the `MyPawn` pointer of the weapon when it is equipped. This is just so it makes things easier later in the code.

Now let's open up `AssaultWeapon`. Add an editable `float` called `FireRate`, and set its default value to `0.05f` in the constructor. Also declare an editable `float` called `WeaponRange` that has a default value of 10000.0f. Finally, declare a `UParticleSystem*` that you'll use for the hit effect (it should be `EditDefaultsOnly`).

Next, create a protected member function called `WeaponTrace`, which will be the function that actually fires a ray. In order to call it every `FireRate` seconds, you can use Unreal's Gameplay Timers.

As for the code in `WeaponTrace`, I've provided (most) of it below:

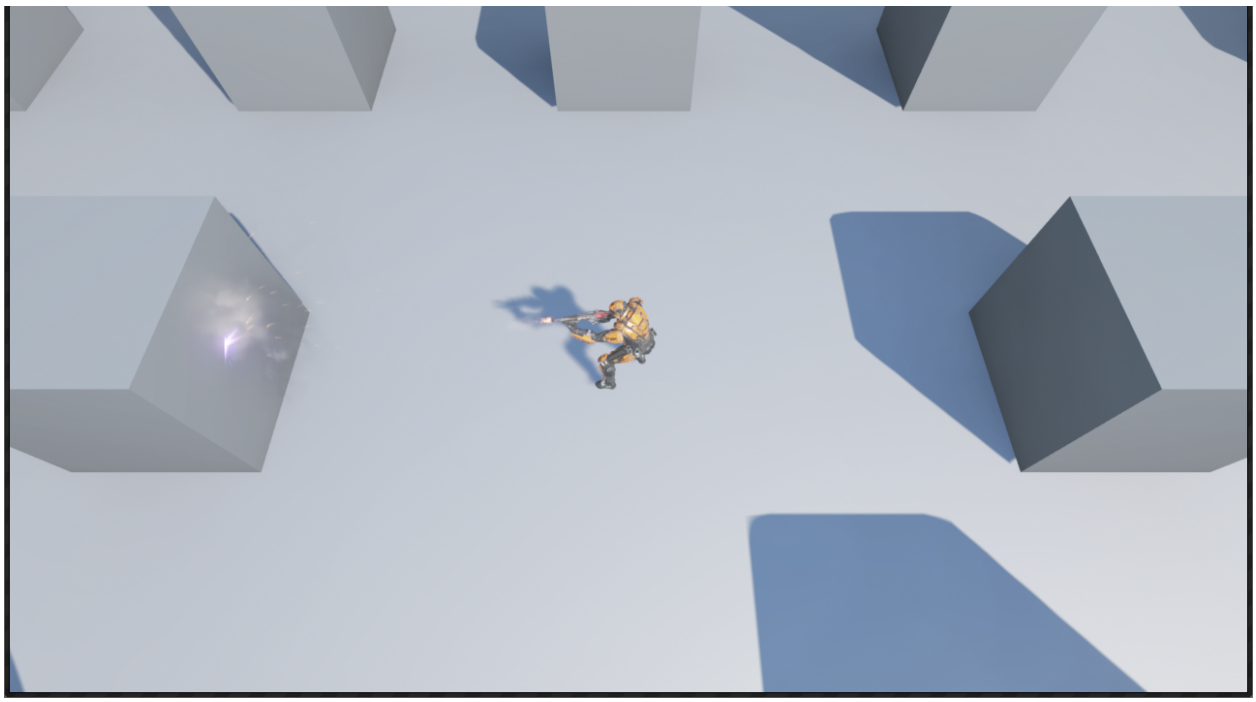
```
void AAssaultWeapon::WeaponTrace()
{
    static FName WeaponFireTag = FName(TEXT("WeaponTrace"));
    static FName MuzzleSocket = FName(TEXT("MuzzleFlashSocket"));

    // Start from the muzzle's position
    FVector StartPos = WeaponMesh->GetSocketLocation(MuzzleSocket);
    // Get forward vector of MyPawn
    FVector Forward = MyPawn->GetActorForwardVector();
    // Calculate end position
    FVector EndPos = /*TODO: Figure out vector math */;


    // Perform trace to retrieve hit info
    FCollisionQueryParams TraceParams(WeaponFireTag, true, Instigator);
    TraceParams.bTraceAsyncScene = true;
    TraceParams.bReturnPhysicalMaterial = true;

    // This fires the ray and checks against all objects w/ collision
    FHitResult Hit(ForceInit);
    GetWorld()->LineTraceSingleByObjectType(Hit, StartPos, EndPos,
        FCollisionObjectQueryParams::AllObjects, TraceParams);

    // Did this hit anything?
    if (Hit.bBlockingHit)
    {
        // TODO: Actually do something
    }
}
```

For the first TODO, you have to use your vector math to calculate the end position using the start position, forward vector, and the weapon's range.

The second TODO is where collision actually needs to be processed. For now, you want to spawn the impact particle effect using `UGameplayStatics::SpawnEmitterAtLocation`.

Once your code compiles, you once again need to update `BP_AssaultWeapon` so that the Impact Effect is `P_AssaultRifle_IH`.

If this all works properly, you'll now be able to shoot at the gray blocks:



Now that the gun fires, let's add something a bit more interesting to shoot at.

Part 3: An Enemy

While we could make enemies that fired guns back at you, it's a bit easier to make a melee enemy, so let's focus on that. Unfortunately, the only rigged melee enemy I found in the samples was the hammer-wielding dwarf from StrategyGame. So we'll use that model! It's easy enough to make a video game story that fits, something like:

The year is 2125. The president has been kidnapped by an army of hammer-wielding dwarves. Are you a bad enough space marine to kill all of the dwarves?

Anyway, at a minimum you need to create two classes in order to have an AI-controlled character in Unreal 4. First, you need class that derives from `Character` which that represents the actual character (in this case, the Dwarf). Then you need a class that derives from `AIController` that represents the behavior that actually controls the character. This is very similar to the delineation with the player character and player controller classes.

For our Dwarf AI, we're going to have very simple behavior. The Dwarf should move towards the player at all times and, upon reaching it, start attacking the player with their hammer. If the Dwarf gets out of range (that is, the player manages to flee the attacking dwarf), the dwarf should stop attacking the player.

Now there's actually a very cool new (and still experimental) feature in Unreal 4 called behavior trees. However, you still need a controller to use them, and they are more complicated than just writing the behavior directly in the `AIController`. They're worthwhile to investigate for the final project, but for our SHMUP we're going to use the simpler solution of coding the behavior directly into the controller.

Just in case we want to add additional enemies/AIs, we're going to add a bit of a hierarchy. Create a C++ class that inherits from `Character` called `EnemyCharacter`, and then make a class that inherits from `EnemyCharacter` called `DwarfCharacter`. Remember that you use "Add code to project" to create these classes.

We want to do the same thing for the controllers: make an `AIEnemyController` class that derives from `AIController` and an `AIDwarfController` class that derives from `AIEnemyController`.

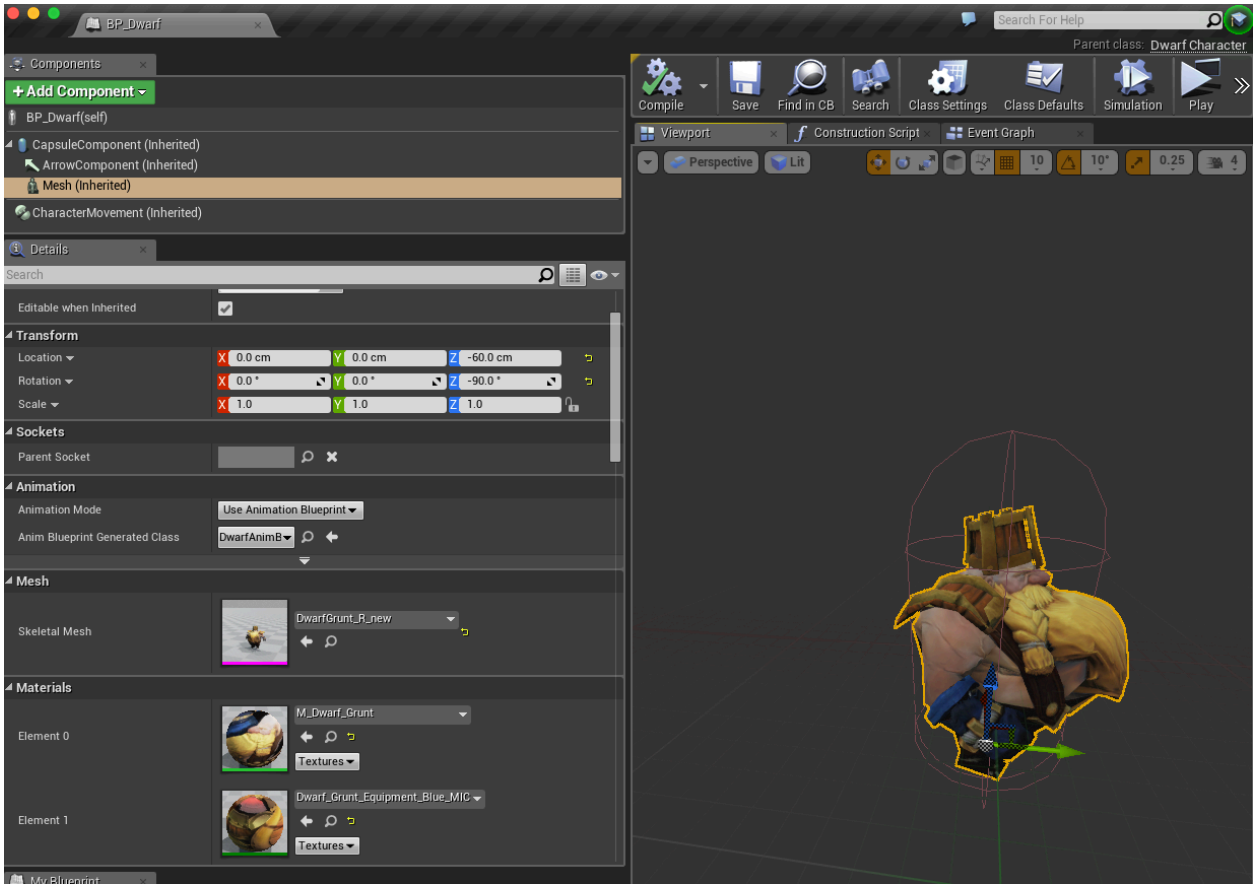
For the moment, we're not going to mess with these new classes. But there is one line of code we need to add to `DwarfCharacter`'s constructor. A character can specify which `AIController` class it should use by default; in our case we want `DwarfCharacter` to automatically use `AIDwarfController`. To do this, add the following line of code:

```
AIControllerClass = AAIDwarfController::StaticClass();
```

Basic Dwarf Blueprint

Now you should be able to build and run the game. In the editor, we want to create a blueprint called `BP_Dwarf` that derives from `DwarfCharacter`. By default, a `Character` has a capsule component and a skeletal mesh component. But we need to tell the component to actually use the dwarf model and animations.

So we need to edit the components `BP_Dwarf`. For the Mesh, set the Skeletal Mesh to `DwarfGrunt_R_new`, the Anim Blueprint to `DwarfAnimBlueprint`, and the Z component of location to `-60`. We also need to set the Z rotation to `-90` so the dwarf is facing down the forward axis (+X in Unreal). Finally, we also need to make the capsule a little bit bigger. So edit the root capsule component and change its radius to `40.0`. Once you make these edits, the `BP_Dwarf` should look like this:

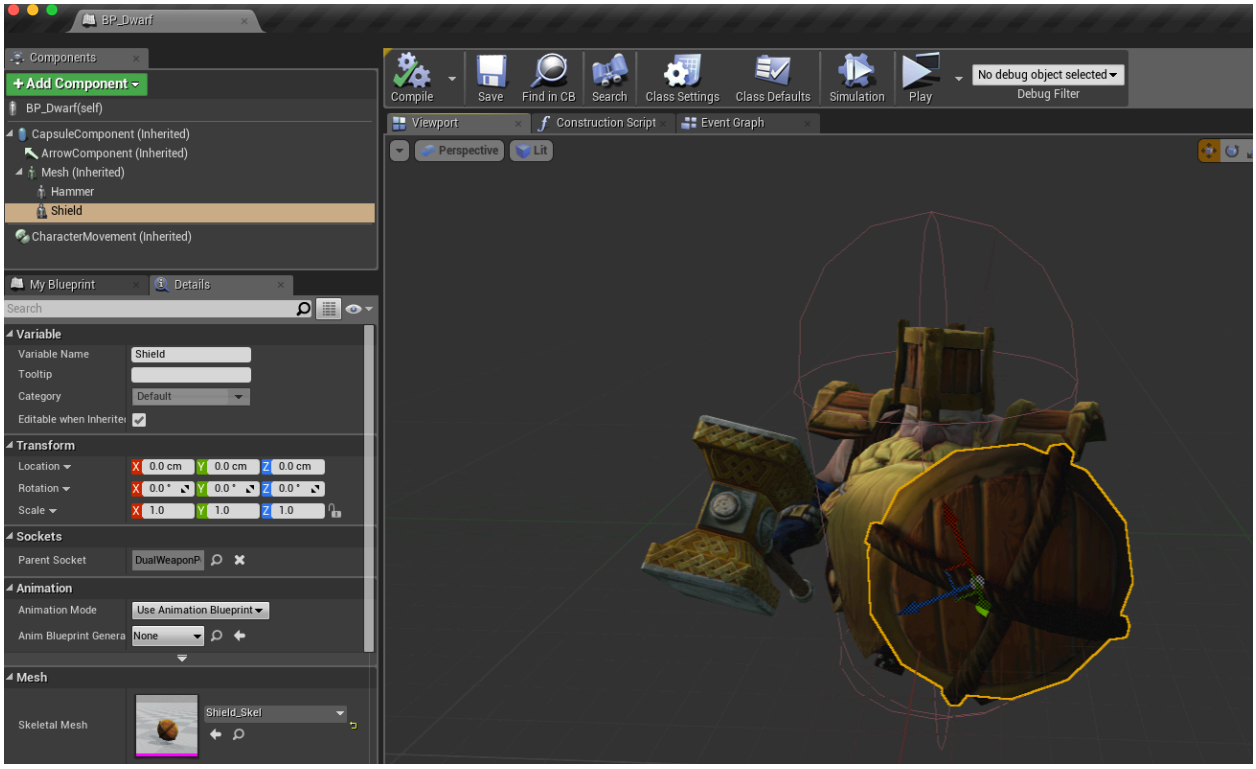


Notice how it's pointing down the character's forward +X axis (even though that corresponds to the model's +Y axis).

However, there's a problem in that the dwarf does not have his trusty hammer and shield! This is because they were created as separate skeletal meshes. We could create new classes to represent these, and equip them much like we did for the assault rifle. However, our game won't allow the player to pick up and use the dwarf's hammer and shield. So to make things simpler, we're just going to add them into `BP_Dwarf` so they're part of the Dwarf by default.

In the component editor, click the Add Component dropdown and select "Skeletal Mesh". Call it `Hammer`. Drag this new mesh onto "Mesh," in the component list, which will add it as a child. You need to then clear out the location/rotation, as these coordinates are now relative to the parent. For the Parent Socket select `WeaponPoint` and for the Skeletal Mesh select `WarHammer`.

Similarly, add another child of "Mesh" called "Shield" that's attached to `DualWeaponPoint` and uses `Shield_Skel`. This should ultimately look something like this:



If you look closely, you'll see that the hammer is actually slightly away from the dwarf's hand – but I double-checked and this appears to just be an error in the skeleton asset.

By the way – if you look at the defaults for `BP_Dwarf`, you should see that its `AIController` class is correctly set to `AIDwarfController`. However, were we to drop a `BP_Dwarf` into the

level, at the moment it wouldn't do anything since the controller has no functionality. So let's fix that.

If you take a look at the documentation for AIController, you'll see that it derives from `Actor`. This means it supports `BeginPlay`, `EndPlay`, and `Tick`. However, there are some other important methods in here. Most notably, there is the `Possess` function, which is called when the `AIController` attaches to a specific `Pawn`. What's confusing is that depending on how the character is spawned, this may be called before or after `BeginPlay`. You'll also notice that there is a very handy `MoveToActor` function as well as `OnMoveCompleted`. Look at the documentation of all of these three functions to understand their functionality. Now let's think about their utility one-by-one.

`Possess`: The idea is to be able to know inside the `AIDwarfController` class the pawn which is being driven (or possessed) by the AI. Recall we associated the `AIDwarfController` as the AI for the dwarf in the `ADwarfCharacter` class? All we want to obtain in the `AIDwarfController` class is the pointer to that dwarf. For that, override the Possess[1] function in `AIDwarfController` class and store the pawn attached to this AI. You may then cast this pawn to `ADwarfCharacter` in order to call functions defined in `ADwarfCharacter` class (for a later step).

`MoveToActor:` What we want to do is override the `BeginPlay` function in the `AIDwarfController` class so that it calls the super `BeginPlay`[1], grabs the player pawn (using the `UGameplayStatics` function you used in the tutorial video in lab07), and calls `MoveToActor` to tell the dwarf to move towards the player. Every parameter other than the first one has a default value, so you really just have to tell it what the target is.

Once you have this code in the game, drop in an instance of `BP_Dwarf` in the level. You'll see that the dwarf paths towards you, and stops moving once he gets right on top of you. However, if you start moving away from the dwarf once it's stopped, it won't move anymore. That's because the move already completed.

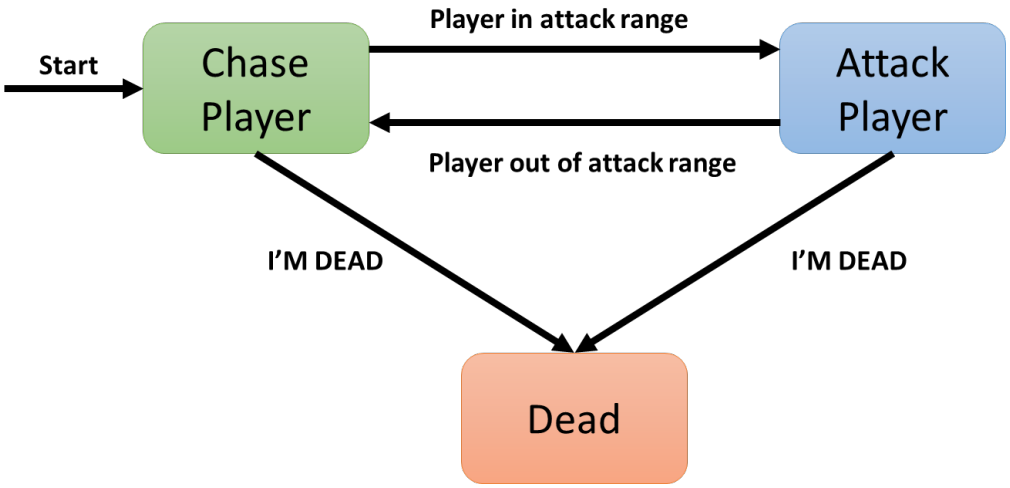
`OnMoveCompleted:` You will need to override `OnMoveCompleted` (with the appropriate parameters) in the `AIDwarfController` class. This function gets called when the dwarf's move to the player is completed. You will need it to implement the complete behavior of the drawf as described next.

---

[1] Typically, when you override a function, you must call the function implementation of its parent (Super) too.

Dwarf State Machine

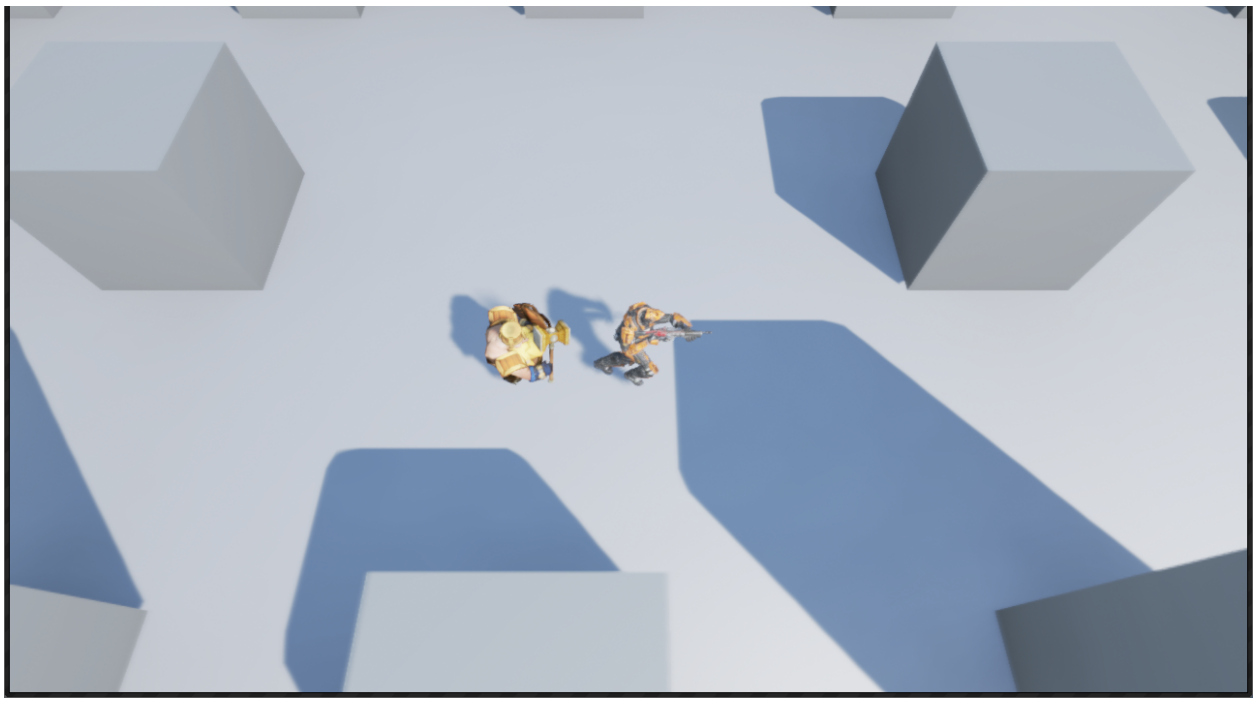
To get the dwarf to continuously move towards the player and attack it, we need to create a state machine inside of `AIDwarfController`. The states should be setup like this:



This can be done in a manner very similar to the game mode state machine that was created in the tutorial video of lab 07. So create an `enum` and a function to change the state. In this `enum`, you will also want to add a "Start" state that is what the dwarf AI is initially set to in `BeginPlay`. Then in `Tick`, if the state is "Start" you'll want to change to the "Chase" state that will start the `MoveTo`. You can then use `OnMoveCompleted` to change to the "Attack" state. The "Attack" state would have to check every frame in `Tick` whether or not the player pawn is still in range of the dwarf pawn (you can add the range as an editable variable in the controller class – set it to `150.0f` by default). When the player gets out of range, the AI should enter the "Chase" state once again.

Don't worry about the actual attack animation and/or doing damage just yet. Just get it working so that the Dwarf always resumes the chase if you get away from him. You can also ignore the "Dead" state for now.

Once this works, you should have the dwarf constantly chasing after you. If for some reason, the dwarf doesn't turn to face the direction he's moving, you may need to check the "orient rotation to movement" box in `BP_Dwarf`.



To play the attack animation, we need to make some changes to `DwarfCharacter`. Add a new `EditDefaultsOnly` property of type `UAnimMontage*` called `AttackAnim`. An anim montage is a sequence of one or more animations that you can invoke via code.

Next, add two void functions into `DwarfCharacter` – `StartAttack` and `StopAttack`. We will call these functions from the `AIDwarfController` when it's time for the dwarf to attack. The implementation of `StartAttack` and `StopAttack` is pretty straightforward, they just need to call `PlayAnimMontage` and `StopAnimMontage`, respectively.

Next, change `AIDwarfController` so when you enter the "Attack" state you call `StartAttack` on the `DwarfCharacter`, and when you exit that state, call `StopAttack`.

Once you get this code in, open up the editor again and assign the `AttackAnim` property in `BP_Dwarf` to `Attack_1_new_Montage`. If everything worked properly, you should now have the dwarf attacking with his hammer.

Applying Damage

Actors have a built-in function called `TakeDamage` that is used to send damage events to them. However, they don't have a built-in concept of hit points or anything like that. So in order to support the dwarf dealing damage to the player (and vice versa), we need to add floats for the

health to both dwarf and player characters. The health variables should be `EditAnywhere`. Similarly, we should add variables to specify how much damage a `Weapon` can do, and how much damage the `DwarfCharacter` can do.

For my defaults, I went with 100 Health Points (HP) for the player, 20 HP for the dwarf, 10 damage for the hammer, and 2 damage for the assault rifle.

Then, we need to override `TakeDamage` for both characters. For example, the `DwarfCharacter` version of `TakeDamage` would look something like this:

```
float ADwarfCharacter::TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,
    AController* EventInstigator, AActor* DamageCauser)
{
    float ActualDamage = Super::TakeDamage(Damage, DamageEvent,
                                           EventInstigator, DamageCauser);

    if (ActualDamage > 0.0f)
    {
        Health -= ActualDamage;
        if (Health <= 0.0f)
        {
            // We're dead
            bCanBeDamaged = false; // Don't allow further damage
            // TODO: Process death
        }
    }

    return ActualDamage;
}
```

Notice how we call the super version of `TakeDamage` first, so it can do all of its normal damage checks to make sure the damage should be happening. If we had planned this out a little bit better, we probably should have made a child of `Character` that both `DwarfCharacter` and the player character inherited from. That way we would only have to override `TakeDamage` once, potentially. As it is, we'll have to override `TakeDamage` in both the dwarf and the player.

The next question is what should happen to the dwarf when it dies? The first thing the code does is sets `bCanBeDamaged` to `false`. This prevents any further damage events from actually doing anything. We should also add in a call to `StopAttack`, which will kill the attack sequence if the dwarf was currently attacking.

The next thing we want to do is play a death animation. As we did for the attack anim, create a new `UAnimMontage*` for the death animation and play the montage when the dwarf dies. One thing that is useful about `PlayAnimMontage` is that it returns the duration of the anim montage it will play – so we can then use a timer to schedule a function that will handle the final destruction

of the character. If we destroy it before the animation is done, you won't see the death animation. (It should be noted that this isn't really the "best" way to handle it – the better approach would be to create a custom anim notify at the exact frame we want the dwarf to be destroyed. But that approach is more complicated.)

The final thing we should do when we detect death in `TakeDamage` is to call `UnPossess` on the controller attached to this dwarf (this can just be accessed based on the `AIController` class variable). The `UnPossess` function tells the controller that it no longer should do updates for this pawn, which will prevent it from switching to the "Chase" or "Attack" states erroneously.

As for the function that is triggered by the timer (once the death animation finishes), the simplest approach is to just immediately call destroy, which will remove the dwarf from the world.

Next, we need to update `AssaultWeapon` so it actually does damage to the dwarf. This should be done in `WeaponTrace`. In the case that the hit was successful, we need to get the actor which was hit and call `TakeDamage` on this actor if it's a dwarf. This will look like:

```
ADwarfCharacter* Dwarf = Cast<ADwarfCharacter>(Hit.GetActor());
if (Dwarf)
{
    Dwarf->TakeDamage(Damage, FDamageEvent(), GetInstigatorController(), this);
}
```

Notice how for the damage event, I'm just passing in an empty `FDamageEvent` – we are not required to give further detail on the damage event (though we certainly can, if we wanted to).

Then in the editor, the `BP_Dwarf` will have to be updated once again to add in the death animation. You want to set it to `Death2_Montage`.

If everything was done correctly, you should now have a dwarf that plays its death animation and then disappears after you shoot at it for a while.

Now let's make the dwarf's hammer attack damage the player. To do this, what you can do is setup a looping timer with the duration of the attack animation montage in the `StartAttack` function. The function you call via the timer should call `TakeDamage` on the player's pawn (you can grab it via `UGameplayStatics`). Don't forget to clear this timer in `StopAttack`!

Then in the player character class, override `TakeDamage` and make it so that when the player's health hits 0, it plays a death animation montage as well. We can also turn on cinematic mode on the player's controller to (in theory) prevent further input.

You'll then want to update the `TopDownCharacter` blueprint to set the death animation to `Death_Montage`. If you run this now, you should see when the player dies, he plays the death animation. However, you'll notice two issues. When the death animation ends, the player gets back up. Furthermore, the player can still use mouse look and fire the gun, which clearly should not be allowed once the player is dead.

To fix the player getting back up, you need to call `Deactivate` on the `Mesh` once the death animation is done. `Deactivate` tells the skeletal mesh to stop updating its animations. So you'll need to get the duration of the animation and trigger a function on a timer, just like you did in the dwarf character's code. The duration of the timer would need to be just a little bit less than the duration of the animation (like 0.25f seconds, so that you are disabling the animation before the character is back on his feet!).

To fix the issue with the mouse look and weapon firing, we first need to add a check in `UpdateMouseLook` in the player controller. We don't want to do the mouse hit or rotation if the controller's `IsLookInputIgnored` returns `true`. For that to work, we will need to set the functions `SetIgnoreLookInput` and `SetIgnoreMoveInput` when player dies. We should then also add a function called `IsDead` to the player character that returns `true` if the player is dead. Then update `OnStartFire` and `OnStopFire` in the player controller to not fire the weapon if `IsDead` returns `true`.

Notice that dwarf still keeps on attacking even after player is dead. Add logic in dwarf's AI controller to check if player is dead and, therefore, stop attack animation.

Part 4: Spawn Manager

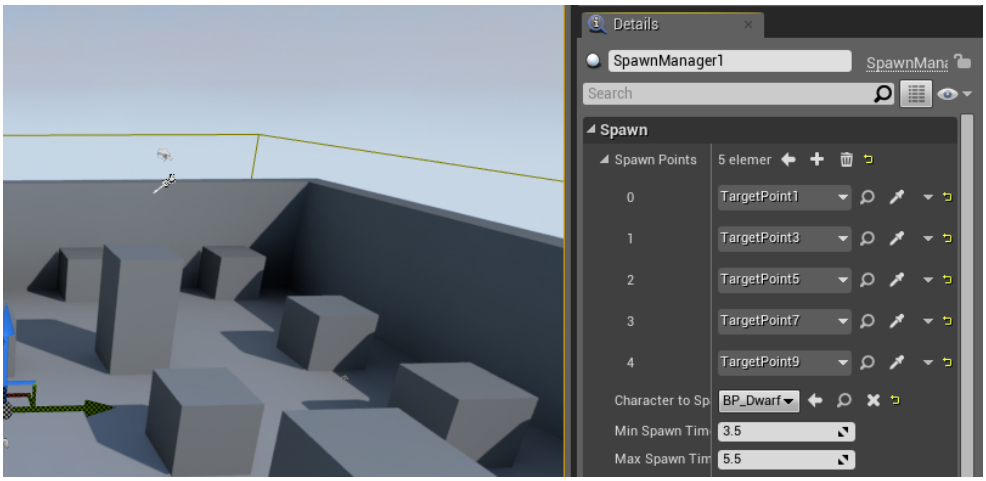
We're now going to create a class to manage the spawning of dwarves. You created a spawning volume in Epic's tutorial, but our spawn manager will be a little different. Rather than picking a location inside the volume, we'll have an array of target locations that we place in the level. The spawn manager will then randomly pick a location from this array and spawn a dwarf there.

Create a new class that inherits from Actor called `SpawnManager`. This class needs to have four `EditAnywhere` properties. It'll need a `TArray` of the `ATargetPoint` pointers and a `TSubclassOf ACharacter`, which will be used for specifying the target points and the character to spawn, respectively. It'll also need two floats for the min spawn time and max spawn time. I added these four properties in the `Spawn` category. They'll have to be set on the spawn manager instance in the level.

In terms of the functions in `SpawnManager`, there are two ways to implement it. The first way is the way that it was done in Epic's tutorial, where you make a standalone float that you subtract from in `Tick`. The other way (which was how I did it), is to utilize timers. So you can override `BeginPlay` to create the initial (non-looping) timer. Then the function that does the spawning will both spawn the character as well as set a new timer to a new random duration between min spawn time and max spawn time.

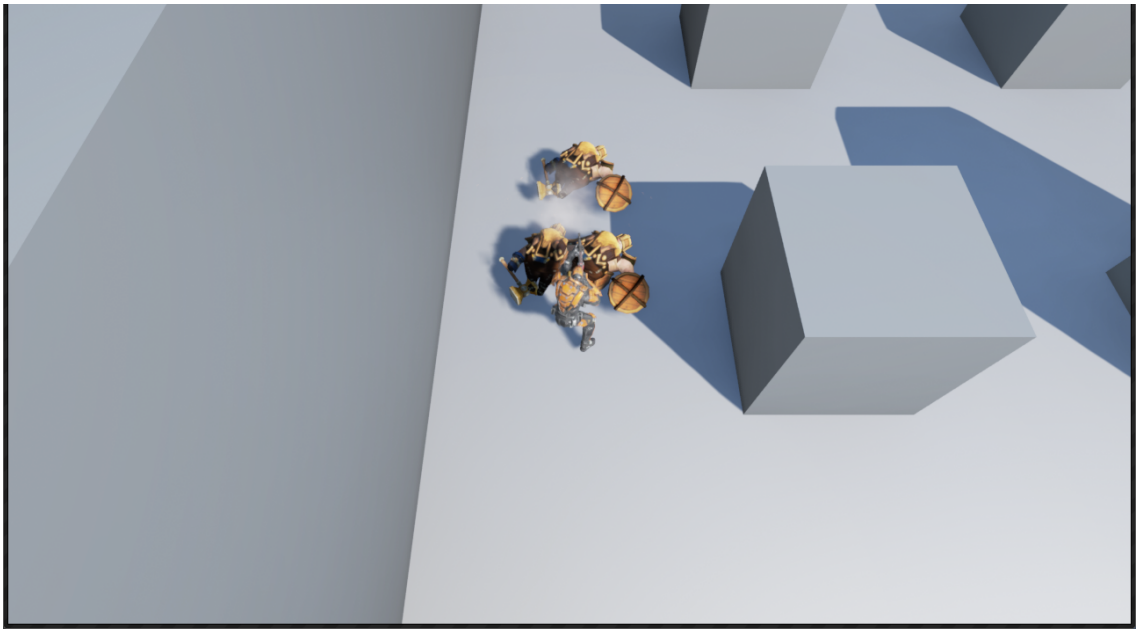
Either way you implement it, the spawning calls will be similar to what we did when spawning the weapon. However, there are two important additions: 1) You have to randomly choose the spawn location to be one of the target points (chosen from the array you set up earlier and which you will populate in the editor), 2) For whatever reason, when spawning a character it does not automatically create its default AI controller. So once you spawn the character via `SpawnActor`, you must call `SpawnDefaultController` on it.

Once you have this code, place target points in the level (wherever you want dwarves to spawn), and also place a `SpawnManager` in the level. You'll need to set the parameters directly on the `SpawnManager` instance in the level:



Once you've got the spawn manager and target points placed in the level, run it and you should see dwarves spawning and chasing after you.

By the way – if you want to improve the avoidance behavior for the dwarves, inside the BP_Dwarf blueprint check the "Use RVOAvoidance" checkbox.



Extra Credit

For extra credit, I want you to take all that you have learned and add something unique to your game. If you don't have any ideas, here are some:

- Add a HUD – a bar to show the player's health, and maybe smaller bars floating above each dwarf.
- Add a rocket launcher – the assets for the rocket launcher are all there, so you could make a new weapon class and maybe a way to give it to the player?
- Add health pickups – Maybe every so often a dwarf drops a health pickup that, if you walk over it, let's you pick it up.

Grading Rubric (when running the code)

Part 1: Movement and Facing	10%
Part 2: Adding a Weapon	30%
Part 3: An Enemy	40%
Part 4: Spawn Manager	20%
Total	**100%**

Remember that this is **10%** of your overall course grade.

You'll get up to 10 extra credit points as long as you have a working(ish) gameplay mechanic that required a reasonable amount of effort.

Demo grade

Part 4 (due on April 09): 2%