Roberto Reynoso

Code Analysis State Pattern Player Movement

GitLab Repo: https://eae-git.eng.utah.edu/u1482733/GetOutOfMyState

How did I implement the State Pattern?

```cpp
class GETOUTOFMYSWAMP_API State
{
public:
    State();
    virtual ~State();
    virtual void OnEnter(AShrek* shrek) = 0;
    virtual void OnUpdate(AShrek* shrek, float DeltaTime) = 0;
    virtual void OnExit(AShrek* shrek) = 0;
};
```

I started with making an abstract State class that I could use to then implement each player specific state that I needed to implement. This class contains three different virtual functions. OnEnter() is called when the player enters that state. OnUpdate() is called every tick while the player is in that state. OnExit() is called when the player leaves that state to enter a new state.

```cpp
class GETOUTOFMYSWAMP_API IdleState : public State
{
public:
    void OnEnter(AShrek* shrek) override
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(
                -1,
                5.f,
                FColor::Yellow,
                FString::Printf(TEXT("Entered Idle State"))
            );
        }
    }

    void OnUpdate(AShrek* shrek, float DeltaTime) override
    {

        shrek->AddActorLocalRotation(FRotator(0.0f, 1.0f, 0.0f));
    }

    void OnExit(AShrek* shrek) override
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(
                -1,
                5.f,
                FColor::Yellow,
                FString::Printf(TEXT("Exited Idle State"))
            );
        }
    }
};
```

The IdleState() won't do anything OnEnter() apart from calling a debug message to print to the screen. OnUpdate() will rotate the player indefinitely in place. OnExit() will just call a debug message when leaving this state to a new state.

```
class GETOUTOFMYSWAMP_API WalkState : public State
{
public:

    FVector currentMovement = FVector::Zero();

    void SetMovement(FVector movementDirection)
    {
        currentMovement = movementDirection;
    }



    void OnEnter(AShrek* shrek) override
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(
                -1,
                5.f,
                FColor::Yellow,
                FString::Printf(TEXT("Entered Walk State"))
            );
        }
    }

    void OnUpdate(AShrek* shrek, float DeltaTime) override
    {
        shrek->currentVelocity = (currentMovement * shrek->maxSpeed);
        shrek->AddMovementInput(shrek->currentVelocity);
    }

    void OnExit(AShrek* shrek) override
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(
                -1,
                5.f,
                FColor::Yellow,
                FString::Printf(TEXT("Exited Walk State"))
            );
        }

        shrek->ConsumeMovementInputVector();
    }
};
```

The WalkState() won't do anything on OnEnter() apart from calling a debug message. OnUpdate() will set Shrek's current velocity and then add the movement input to the character object of Shrek to move him either left or right. OnExit() will a function that will just make sure that Shrek's velocity is zero, so that he will stop moving.

```cpp
class GETOUTOFMYSWAMP_API JumpState : public State
{
public:
    void OnEnter(AShrek* shrek) override
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(
                -1,
                5.f,
                FColor::Yellow,
                FString::Printf(TEXT("Entered Jump State"))
            );
        }

        if (shrek->GetCharacterMovement()->IsMovingOnGround())
        {
            shrek->isJumping = false;
        }

        if (!shrek->isJumping)
        {
            shrek->isJumping = true;
            shrek->soundManager->PlaySound(shrek->soundManager->helloThere, shrek->GetActorLocation());
            shrek->LaunchCharacter(FVector(0, 0, 500.0f), true, true);
        }

    }

    void OnUpdate(AShrek* shrek, float DeltaTime) override
    {
        if (shrek->GetCharacterMovement()->IsMovingOnGround())
        {
            shrek->currentState = shrek->playerStates->idleState;
        }
    }

    void OnExit(AShrek* shrek) override
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(
                -1,
                5.f,
                FColor::Yellow,
                FString::Printf(TEXT("Exited Jump State"))
            );
        }
    }
};
```

The JumpState(), within the OnEnter() function will make sure that to check whether or not Shrek is on the ground and whether or not isJumping is set to false. If isJumping is false Shrek will then jump using the LaunchCharacter() function. OnUpdate() will check whether or not Shrek has landed on the ground if Shrek is has landed, Shrek's state will change to the IdleState(). OnExit() only calls a debug message.

```cpp
class GETOUTOFMYSWAMP_API Command
{
public:
    Command();
    virtual ~Command();
    virtual void Execute() = 0;
    AShrek* shrek;

};
```

Interesting fact about my project is that I decided to combine the Command Pattern with the State Pattern.

```
class GETOUTOFMYSWAMP_API MoveLeftCommand : public Command
{
public:
    void Execute() override
    {
        shrek->currentState->OnExit(shrek);
        shrek->playerStates->walkState->SetMovement(FVector(-1.0f, 0.0f, 0.0f));
        shrek->currentState = shrek->playerStates->walkState;
        shrek->currentState->OnEnter(shrek);

    };
};

class GETOUTOFMYSWAMP_API MoveRightCommand : public Command
{
public:
    void Execute() override
    {
        shrek->currentState->OnExit(shrek);
        shrek->playerStates->walkState->SetMovement(FVector(1.0f, 0.0f, 0.0f));
        shrek->currentState = shrek->playerStates->walkState;
        shrek->currentState->OnEnter(shrek);
    };
};

class GETOUTOFMYSWAMP_API IdleCommand : public Command
{
public:
    void Execute() override
    {
        shrek->currentState->OnExit(shrek);
        shrek->currentState = shrek->playerStates->idleState;
        shrek->currentState->OnEnter(shrek);
    };
};


class GETOUTOFMYSWAMP_API JumpCommand : public Command
{
public:
    void Execute() override
    {
        shrek->currentState->OnExit(shrek);
        shrek->currentState = shrek->playerStates->jumpState;

        if (shrek)
        shrek->currentState->OnEnter(shrek);
    };
};
```

Here you can see how each state calls it's OnExit() and OnEnter() functions. Each state is called based on the command that is executed. I do find this structure quite nice, since all of my commands and states are separated in a way that is extremely easy to read. This system is very flexible where it would be very easy to add another Command or State based on the needs that the game might call for.

```
class State;
class Command;

UCLASS()
0 derived Blueprint classes
class GETOUTOFMYSWAMP_API AShrek : public ACharacter
{
    GENERATED_BODY()

public:
    // Sets default values for this pawn's properties
    AShrek();

    State* currentState = nullptr;

    Command
        * aKey,
        * dKey,
        * spacebar,
        * noKey;

    bool isJumping = false;

    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Sound Manager")
    Changed in 0 Blueprints
    ASoundManager* soundManager;

    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Player State Manager")
    Changed in 0 Blueprints
    APlayerStateManager* playerStates;

    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Speed")
    Changed in 0 Blueprints
    float maxSpeed = 500;

    FVector currentVelocity = FVector::Zero();

    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input Mapping")
    Changed in 0 Blueprints
    class UInputMappingContext* InputMapping;

    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input")
    Changed in 0 Blueprints
    TObjectPtr<UInputAction> MoveAction;

    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input")
    Changed in 0 Blueprints
    TObjectPtr<UInputAction> JumpAction;
```

Here I would like to just show off what the Header file looks like for Shrek. Everything here is quite readable in my opinion. The two Object Types to look at are the State and Command variables, these work very well together while being their own self-contained systems, meaning one can work just fine without the other.

```
// Sets default values
AShrek::AShrek()
{
    // Set this pawn to call Tick() every frame.  You can turn this off to improve performance if you don't ne
    PrimaryActorTick.bCanEverTick = true;

    aKey = nullptr;
    dKey = nullptr;
    spacebar = nullptr;
    noKey = nullptr;
}

// Called when the game starts or when spawned
void AShrek::BeginPlay()
{
    Super::BeginPlay();

    aKey = new MoveLeftCommand();
    aKey->shrek = this;
    dKey = new MoveRightCommand();
    dKey->shrek = this;
    spacebar = new JumpCommand();
    spacebar->shrek = this;
    noKey = new IdleCommand();
    noKey->shrek = this;

    currentState = playerStates->idleState;
}

// Called every frame
void AShrek::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (currentState != nullptr)
    {
        currentState->OnUpdate(this, DeltaTime);
    }

}
```

Here we can see how the commands and the Shrek's current state variables are set up on BeginPlay(). Within the Tick() function we see that the state's OnUpdate() function is called for whatever the current state that Shrek is on.

```
void AShrek::Move(const FInputActionValue& Value)
{
    FVector movementInput = Value.Get<FVector>();

    if (movementInput.X < 0) // Left
    {
        aKey->Execute();
    }
    else if (movementInput.X > 0) // Right
    {
        dKey->Execute();
    }
    else if (movementInput.X == 0)
    {
        noKey->Execute();
    }
}

void AShrek::Jump(const FInputActionValue& Value)
{
    if (Value.Get<bool>())
    {
        spacebar->Execute();
    }
}
```

These here are where the command Execute() functions are called depending on the action that the player performs.

I was able to learn quite a bit within this project. I decided that I only had the time to basically check off a box this time given the time I had over the week. Though I am happy with how I combined both my Command system with my State Machine system. Though I didn't implement a hierarchy or concurrent state machine mainly due to time constraints, but also the project that I decided to make didn't really call for it. Though if I did have the time, it could have been cool to play around with the idea of either system and see what I could come up with in terms of applying it to a game project.

This project overall was about 10+ hours in terms of time spent working on the project. I ran into a handful of problems, though this was mainly due to unreal and not really how I decided to structure out my code base. Unreal has a lot of presets by default, which shouldn't be the case since they actually prevent some movement options if you decided to use a character object with physics. I had to change the presets to get my player moving. Along with that I had to use special character-based functions to get my player to jump. Having to learn Unreal Engine's specific rules was quite annoying especially because the documentation online wasn't very helpful when trying to solve these issues. Initially I was using a Pawn Object as my player object, since I didn't want all of the features that the Character Object came with, but for some reason using Unreal Engine's physis system just was not playing nice for some reason on the Pawn object while using C++. I would like to continue to look into this issue, but due to time constraints I decided to work with a Character Object instead. The funny part is within an online thread, some people were dealing with a similar issue and the top comment said quote "just use a Character Object". This in itself is so annoying because I would like to work with a simple Object that I can turn into a player object without all of the add on features that the Character Object provides and sometimes forces you to use.

Overall, the project was a success and I am happy with what I was able to take away from this assignment.