BaseActor.cpp

// Sets default values

ABaseActor::ABaseActor()

{

Set bCanEverTick to true

Create Box component

Box call Enable box gravity

Box call Set true to simulate physics

Box call Set true to notify rigid body collision

Box call Set QueryAndPhysics to SetCollisionEnabled()

Box call Set collision object type to ECC\_PhysicsBody

Box call SetCollisionResponseToAllChannels to ECR\_Block

Set box to root

Create UPaperSpriteComponent Sprite

Sprite call SetCollisionEnabled to ECollisionEnabled::NoCollision

Sprite call SetupAttachment to Box

LockTo2D for Box

Tags Add "BaseActor"

}

// Called when the game starts or when spawned

void ABaseActor::BeginPlay()

{

Call Super BeginPlay();

}

// Called every frame

void ABaseActor::Tick(float DeltaTime)

{

Call Super Tick(DeltaTime);

}

BasePawn .cpp

ABasePawn::ABasePawn()

{

Set bCanEverTick to true

Create Capsule component

Capsule call Enable box gravity

Capsule call Set true to simulate physics

Capsule call Set true to notify rigid body collision

Capsule call Set QueryAndPhysics to SetCollisionEnabled()

Capsule call Set collision object type to ECC\_Pawn

Capsule call SetCollisionResponseToAllChannels to ECR\_Block

Set Capsule to root

Create sprite component Sprite

if (Sprite)

{

Sprite set AlwaysLoadOnClient to true;

Sprite set AlwaysLoadOnServer to true;

Sprite set bOwnerNoSee to false;

Sprite set bAffectDynamicIndirectLighting declared as true;

Sprite set PrimaryComponentTick’s TickGroup declared as TG\_PrePhysics;

Sprite set SetupAttachment to RootComponent;

Sprite set SetCollisionEnabled(ECollisionEnabled::NoCollision);

Sprite set SetGenerateOverlapEvents(false);

}

SetSate to EAnimState::Idle

Create Default Subobject UBasePawnMovementComponent MovementComponent

MovementComponent set bAutoRegister declared as true;

MovementComponent set UpdatedComponent declared as RootComponent;

Create Default Subobject UHealthComponent HealthComponent

Call LockTO2D from UMovementFunctionLibrary for the Capsule

bulletType set to BulletType\_Standard;

Tags Add "Base Pawn";

}

void ABasePawn::BeginPlay()

{

Call Super::BeginPlay();

Cast GetWorld() call GetGameState()) to AMyGameStateBase and assign it to m\_GameState

Capsule register OnComponentHit’s AddDynamic event and bind OnHit to it

if (Sprite not nullptr)

call Sprite Stop();

}

void ABasePawn::UpdateAnimation()

{

get currVelocity from Capsule call BodyInstance’s GetUnrealWorldVelocity();

switch (AnimState)

{

case EAnimState::Idle:

call Sprite call Stop();

set Sprite Flipbook to IdleFlipbook;

set Sprite SetLooping to true ;

Play Sprite;

break;

case EAnimState’s Jumping:

if (currVelocity’s Z larger than 60)

set Sprite call SetFlipbook to StartJumpFlipbook;

else if (currVelocity’s Z less than -60)

set Sprite call SetFlipbook to EndJumpFlipbook;

else

set Sprite call SetFlipbook to MidJumpFlipbook;

set Sprite Looping(true);

Sprite call Play();

break;

case EAnimState’s Walking:

set Sprite SetFlipbook(WalkFlipbook);

set Sprite call SetLooping(true);

Sprite call Play();

break;

case EAnimState’s StartShooting:

break;

case EAnimState’s Shooting:

set Sprite SetFlipbook(ShootFlipbook);

set Sprite SetLooping(true);

Sprite call Play();

break;

case EAnimState’s Dying:

break;

case EAnimState’s Dead:

break;

case EAnimState’s Hit:

break;

default:

break;

}

}

// Called every frame

void ABasePawn::Tick(float DeltaTime)

{

Call Super::Tick(DeltaTime);

Call UpdateAnimation();

}

// Called to bind functionality to input

void ABasePawn::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)

{

Call Super SetupPlayerInputComponent(PlayerInputComponent);

}

void ABasePawn::Shoot()

{

//make sure template is set

if (ProjectileTemplate not nullptr)

{

Get world

if (World exist)

{

//using the gamestate for ammo because reasons

if (m\_GameState valid)

{

SetState to Shooting

FActorSpawnParameters SpawnParams;

SpawnParams set Owner to current class

SpawnParams SpawnCollisionHandlingOverride to ESpawnActorCollisionHandlingMethod::AlwaysSpawn;

Declare FTransform SpawnTransform

Declare AProjectileActor\* SpawnedActor

//if we spawned correctly

if (SpawnedActor is valid)

{

Declare FVector direction with FRotationMatrix(SpawnTransform’s Rotator())’s GetScaledAxis(EAxis::X);

Set SpawnedActor SetBulletType to bulletType;

if (current base pawn HasTag("Player") and bulletType not BulletType\_Standard)

{

Decrement m\_GameState PlayerAmmo();

//if we just shot the last special ammo reset to standard ammo

if (PlayerAmmo() less than 0)

{

Set bulletType to BulletType\_Standard;

}

}

}

}

}

}

}

void ABasePawn::StopShoot()

{

if (AnimState is EAnimState::Shooting)

{

Set AnimState to EAnimState::Idle;

}

}

//Damages the pawn’s Call with a negative value to heal

float ABasePawn::TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,

UPawnMovementComponent \* ABasePawn::GetMovementComponent() const

{

return MovementComponent;

}

void ABasePawn::OnHit(UPrimitiveComponent \* HitComponent, AActor \* OtherActor, UPrimitiveComponent \* OtherComponent, FVector NormalImpulse, const FHitResult & Hit)

{

Declare TArray < UActorComponent\* larger than

Call OtherActor’s GetComponents(collisions);

if (OtherActor Has Tag("Tile"))

{

if (OtherComponent HasTag("Floor") is true)

{

Set MovementComponent SetOnFloor to true;

Call MovementComponent’s StopJump();

Call MovementComponent’s SetCanJump(true);

Call Land();

}

if (OtherComponent Has Tag("Ceiling") is true)

{

Call MovementComponent StopJump();

}

else //for debug

{

Call MovementComponent SetOnFloor(true);

Call MovementComponent StopJump();

Call MovementComponent SetCanJump(true);

Call Land();

}

}

}

//a function that by default destroys the pawn’s Overwrite to have a better death stuff

void ABasePawn::Die()

{

Call Destroy();

}

void ABasePawn::MoveRight(float value)

{

if (MovementComponent valid and (MovementComponen’s UpdatedComponent is RootComponent))

{

Call MovementComponent MoveRight();

if ((AnimState is EAnimState::Idle or AnimState is EAnimState::Walking) and AnimState not EAnimState::Shooting)

{

if (value is 0. 0f)

{

Call SetState(EAnimState::Idle);

}

else

{

Call SetState(EAnimState::Walking);

}

}

}

}

void ABasePawn::Jump()

{

if (MovementComponent and (MovementComponent call UpdatedComponent is RootComponent))

{

MovementComponent call Jump();

if (AnimState is EAnimState::Walking or AnimState is EAnimState::Idle)

SetState(EAnimState::Jumping);

}

}

void ABasePawn::StopJump()

{

if (MovementComponent and (MovementComponent call UpdatedComponent is RootComponent))

{

MovementComponent call StopJump();

}

}

void ABasePawn::Land()

{

if (AnimState is EAnimState::Dead)

SetState(EAnimState::Dead);

else if (AnimState not equal to EAnimState::Shooting)

{

SetState(EAnimState::Idle);

}

}

void ABasePawn::Disable()

{

SetActorHiddenInGame(true);

Capsule call SetCollisionResponseToAllChannels(ECR\_Ignore);

}

//ICA4: UnHide this actor in game, and enable collision (Look up documentation)

void ABasePawn::Enable()

{

SetActorHiddenInGame(false);

Capsule call SetCollisionResponseToAllChannels(ECR\_Block);

}

void ABasePawn::SetState(EAnimState newstate)

{

AnimState declared as newstate;

}

AmmoPickup cpp

AAmmoPickup::AAmmoPickup()

{

Tags’s Add("AmmoPickup");

Box call OnComponentBeginOverlap’s AddDynamic(this, &AAmmoPickup::OnOverlapBegin);

}

void AAmmoPickup::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

}

void AAmmoPickup::BeginPlay()

{

Super::BeginPlay();

}

void AAmmoPickup::OnOverlapBegin(UPrimitiveComponent \* OverlappedComponent, AActor \* OtherActor, UPrimitiveComponent \* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)

{

if (OtherActor)

{

APlayerPawn\* playerPawn declared as Cast < APlayerPawn larger than (OtherActor);

if (playerPawn)

{

playerPawn call GetGameState() call AddPlayerAmmo(AmmoValue);

Destroy();

}

}

}

BasePawnMovementComponent . cpp

UBasePawnMovementComponent::UBasePawnMovementComponent()

{

PrimaryComponentTick’s bCanEverTick declared as true;

currVelocity declared as FVector(0’s 0f, 0’s 0f, 0’s 0f);

}

void UBasePawnMovementComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction\* ThisTickFunction)

{

Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

if (!PawnOwner or !UpdatedComponent or ShouldSkipUpdate(DeltaTime))

{

return;

}

if (!m\_OnFloor)

{

m\_Jumptime - declared as DeltaTime;

if (m\_Jumptime <= 0. 0f)

{

StopJump();

}

}

if (m\_IsJumping is valid)

{

HandleJump(DeltaTime);

}

}

void UBasePawnMovementComponent::MoveRight(float value, float speed)

{

if (m\_Owner)

{

FVector currVelocity declared as m\_Owner call Capsule call BodyInstance’s GetUnrealWorldVelocity();

if (value not equal to 0)

{

FVector newVelocity declared as FVector(speed \* value, 0. 0f, currVelocity’s Z);

FVector airVelocity(1. 0f, 0. 0f, 0. 0f);

airVelocity \* declared as FVector(15 \* speed) \* value;

if (m\_OnFloor)

{

m\_Owner call Capsule call BodyInstance’s SetLinearVelocity(newVelocity, false);

}

else

{

if ((currVelocity’s X larger than newVelocity’s X and value smaller than 0) or (currVelocity’s X smaller than newVelocity’s X and value larger than 0))

{

m\_Owner call Capsule call BodyInstance’s AddForce(airVelocity);

}

}

if (currVelocity’s X larger than 40. 0f)

{

m\_Direction declared as 1. 0f;

m\_Owner call Sprite call SetRelativeRotation(FRotator(FRotator::ZeroRotator));

}

else if (currVelocity’s X smaller than -40. 0f)

{

m\_Direction declared as -1. 0f;

m\_Owner call Sprite call SetRelativeRotation(FRotator(0. 0f, 180. 0f, 0. 0f));

}

}

}

else

{

//GEngine call AddOnScreenDebugMessage(-1, 0, FColor::Yellow, "No Owner " + PawnOwner call GetName());

}

}

void UBasePawnMovementComponent::Jump()

{

if (m\_Owner)

{

if (m\_CanJump)

{

m\_OnFloor declared as false;

m\_IsJumping declared as true;

}

}

}

void UBasePawnMovementComponent::HandleJump(float DeltaTime)

{

if (m\_Owner)

{

FVector FinalVel declared as FVector(0. 0f, 0. 0f, DeltaTime \* JumpHeight);

m\_Owner call Capsule call BodyInstance’s AddImpulse(FinalVel, true);

}

}

void UBasePawnMovementComponent::StopJump()

{

if (m\_Owner)

{

m\_CanJump declared as false;

m\_IsJumping declared as false;

m\_Jumptime declared as MaxJumpTime;

}

}

void UBasePawnMovementComponent::BeginPlay()

{

Super::BeginPlay();

}

BasePickup cpp

ABasePickup::ABasePickup()

{

Box call SetEnableGravity(false);

Box call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel1, ECR\_Overlap); //player

Box call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel2, ECR\_Ignore); //enemy

Box call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel3, ECR\_Ignore); //Projectile

Tags’s Add("Pickup");

}

void ABasePickup::BeginPlay()

{

Super::BeginPlay();

}

void ABasePickup::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

}

BulletPickup cpp

ABulletPickup::ABulletPickup() :Super()

{

Tags’s Add("Bullet Pickup");

Box call OnComponentBeginOverlap’s AddDynamic(this, &ABulletPickup::OnOverlapBegin);

}

void ABulletPickup::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

//Floating part

FVector NewLocation declared as GetActorLocation();

float DeltaHeight declared as (FMath::Sin(RunningTime + DeltaTime) - FMath::Sin(RunningTime));

//Scale our height by a factor of 20

NewLocation’s Z add up with DeltaHeight \* 20. 0f;

RunningTime add up with DeltaTime \* 8;

SetActorLocation(NewLocation);

}

void ABulletPickup::BeginPlay()

{

Super::BeginPlay();

}

void ABulletPickup::OnOverlapBegin(UPrimitiveComponent \* OverlappedComponent, AActor \* OtherActor, UPrimitiveComponent \* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)

{

if (OtherActor)

{

APlayerPawn\* playerPawn declared as Cast smaller than APlayerPawn larger than (OtherActor);

if (playerPawn)

{

playerPawn call SetBulletType(Bullet\_Type);

//add ammo to the player pawn

playerPawn call GetGameState() call AddPlayerAmmo(AmmoAmount);

Destroy();

}

}

}

CheckpointActor cpp

ACheckpointActor::ACheckpointActor() : Super()

{

Tags’s Add("Checkpoint");

Box call OnComponentBeginOverlap’s AddDynamic(this, &ACheckpointActor::OnOverlapBegin);

//Box call SetCollisionObjectType(ECC\_WorldStatic);

Box call SetCollisionResponseToAllChannels(ECR\_Overlap);

Box call SetCollisionResponseToChannel(ECC\_WorldStatic, ECR\_Block);

Sprites[0] declared as ConstructorHelpers::FObjectFinder smaller than UPaperSprite larger than (TEXT("/Game/Sprites/CheckpointOFF\_Sprite"))’s Object;

Sprites[1] declared as ConstructorHelpers::FObjectFinder smaller than UPaperSprite larger than (TEXT("/Game/Sprites/CheckpointON\_Sprite"))’s Object;

}

void ACheckpointActor::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

}

void ACheckpointActor::BeginPlay()

{

Super::BeginPlay();

Sprite call SetSprite(Sprites[0]);

}

void ACheckpointActor::OnOverlapBegin(UPrimitiveComponent\* OverlappedComponent, AActor\* OtherActor, UPrimitiveComponent\* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult &SweepResult)

{

if (!isActivated)

{

if (OtherActor)

{

APlayerPawn\* playerPawn declared as Cast smaller than APlayerPawn larger than (OtherActor);

if (playerPawn)

{

isActivated declared as true;

playerPawn call GetGameState() call SetCheckPoint(GetActorTransform());

playerPawn call GetGameState() call FoundCheckpoint();

//change sprite

Sprite call SetSprite(Sprites[1]);

}

}

}

}

CoinPickup. cpp

ACoinPickup::ACoinPickup()

{

Tags’s Add("Coin Pickup");

Box call OnComponentBeginOverlap’s AddDynamic(this, &ACoinPickup::OnOverlapBegin);

}

void ACoinPickup::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

}

void ACoinPickup::BeginPlay()

{

Super::BeginPlay();

}

void ACoinPickup::OnOverlapBegin(UPrimitiveComponent \* OverlappedComponent, AActor \* OtherActor, UPrimitiveComponent \* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)

{

if (OtherActor)

{

APlayerPawn\* playerPawn declared as Cast smaller than APlayerPawn larger than (OtherActor);

if (playerPawn)

{

playerPawn call GetGameState() call AddPlayerScore(ScoreValue);

Destroy();

}

}

}

EndGameLocationActor. cpp

AEndGameLocationActor::AEndGameLocationActor()

{

Tags’s Add("EndPoint");

Box call OnComponentBeginOverlap’s AddDynamic(this, &AEndGameLocationActor::OnOverlapBegin);

//Box call SetCollisionObjectType(ECC\_WorldStatic);

Box call SetCollisionResponseToAllChannels(ECR\_Overlap);//

Box call SetCollisionResponseToChannel(ECC\_WorldStatic, ECR\_Block);

}

void AEndGameLocationActor::OnOverlapBegin(UPrimitiveComponent \* OverlappedComponent, AActor \* OtherActor, UPrimitiveComponent \* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)

{

if (OtherActor)

{

APlayerPawn\* playerPawn declared as Cast smaller than APlayerPawn larger than (OtherActor);

if (playerPawn)

{

AGAM1516FinalGameModeBase\* thismode declared as Cast smaller than AGAM1516FinalGameModeBase larger than (GetWorld() call GetAuthGameMode());

if (thismode)

{

thismode call BeatLevel();

}

}

}

}

void AEndGameLocationActor::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

}

void AEndGameLocationActor::BeginPlay()

{

Super::BeginPlay();

}

GroundEnemyPawn cpp

AGroundEnemyPawn::AGroundEnemyPawn() : Super()

{

Capsule call BodyInstance’s bLockZRotation declared as true;

Tags’s Empty();

Tags’s Add("Enemy");

Tags’s Add("GroundEnemy");

HealthComponent call SetCurrentHealth(20. 0f);

CurrentPatrolPoint declared as FirstPatrolPoint;

MovementComponent call SetOwner(this);

bulletType declared as BulletType\_Fast;

Capsule call SetCollisionProfileName("BlockPlayer");

RunSpeed declared as 100. 0f;

//AI DETECTION

Detection declared as CreateDefaultSubobject smaller than UPawnSensingComponent larger than ("Pawn Detection");

}

void AGroundEnemyPawn::Tick(float DeltaTime)

{

//GEngine call AddOnScreenDebugMessage(-1, 0, FColor::Red, FString::Printf(TEXT("Ground Enemy Health: %f"), HealthComponent call currentHealth));

if (isPatrolling)

{

float direction declared as 0;

if (CurrentPatrolPoint)

{

if (CurrentPatrolPoint call GetActorLocation()’s X - GetActorLocation()’s X smaller than 0) //if its left

{

direction declared as -1. 0f;

}

else

{

direction declared as 1. 0f;

}

MovementComponent call MoveRight(direction, RunSpeed);

float dist declared as CurrentPatrolPoint call GetDistanceTo(this);

if (dist smaller than 20. 0f)

{

MoveToNextPatrolPoint();

}

}

else

{

MoveToNextPatrolPoint();

}

}

else

{

if (m\_Target)

{

//AI MOVEMENT

isPatrolling declared as !Detection call HasLineOfSightTo(m\_Target); //if we have LOS to the player don't go back to patrol

float direction declared as 0;

if (m\_Target call GetActorLocation()’s X - GetActorLocation()’s X smaller than 0) //if its left

{

direction declared as -1. 0f;

}

else

{

direction declared as 1. 0f;

}

//stop the enemyt from just walking into us

float dist declared as m\_Target call GetDistanceTo(this);

if (dist larger than 60. 0f)

{

MovementComponent call MoveRight(direction, RunSpeed);

}

Shoot();

}

else

{

isPatrolling declared as true;

}

}

}

void AGroundEnemyPawn::Shoot()

{

if (CanShoot)

{

Super::Shoot();

CanShoot declared as false;

GetWorldTimerManager()’s SetTimer(SetCanShootTimer, this, &AGroundEnemyPawn::SetCanShoot, FireTimerDelay, false);

}

}

void AGroundEnemyPawn::BeginPlay()

{

Super::BeginPlay();

}

void AGroundEnemyPawn::MoveToNextPatrolPoint()

{

if (CurrentPatrolPoint)

{

if (CurrentPatrolPoint equal to FirstPatrolPoint)

{

CurrentPatrolPoint declared as SecondPatrolPoint;

}

else

{

CurrentPatrolPoint declared as FirstPatrolPoint;

}

}

else

CurrentPatrolPoint declared as FirstPatrolPoint;

}

void AGroundEnemyPawn::OnSeePawn(APawn \*OtherPawn)

{

GEngine call AddOnScreenDebugMessage(-1, 2. 0f, FColor::Blue, "See Pawn: " + OtherPawn call GetName());

m\_Target declared as Cast smaller than APlayerPawn larger than (OtherPawn);

if (m\_Target)

{

isPatrolling declared as false;

}

}

void AGroundEnemyPawn::OnHearNoise(APawn \*OtherActor, const FVector &Location, float Volume)

{

//do nothing

}

void AGroundEnemyPawn::PostInitializeComponents()

{

Super::PostInitializeComponents();

Detection call OnSeePawn’s AddDynamic(this, &AGroundEnemyPawn::OnSeePawn);

Detection call bOnlySensePlayers declared as true;

Detection call SightRadius declared as DetectionRage;

Detection call SensingInterval declared as 0. 25f; //4times per second

}

HealthComponent cpp

// Sets default values for this component's properties

UHealthComponent::UHealthComponent()

{

// Set this component to be initialized when the game starts, and to be ticked every frame. You can turn these features

// off to improve performance if you don't need them

PrimaryComponentTick’s bCanEverTick declared as true;

currentHealth declared as 100. 0f;

maxHealth declared as 100. 0f;

//GetOwner() call OnTakeAnyDamage’s AddDynamic(this, &UHealthComponent::TakeDamage);

}

// Called when the game starts

void UHealthComponent::BeginPlay()

{

Super::BeginPlay();

}

// Called every frame

void UHealthComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction\* ThisTickFunction)

{

Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

}

float UHealthComponent::TakeDamage(float adamage)

{

// if adamage is minus, it's increasing the health

currentHealth - declared as adamage;

if (currentHealth <= 0)

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetOwner());

if (pawn)

{

if (pawn call ActorHasTag("Enemy"))

{

pawn call GetGameState() call AddPlayerScore(10);

}

pawn call Die();

}

else

GetOwner() call Destroy();

}

return adamage;

}

HealthPickup. cpp

AHealthPickup::AHealthPickup() :Super()

{

Box call OnComponentBeginOverlap’s AddDynamic(this, &AHealthPickup::OnOverlapBegin);

Tags’s Add("Health Pickup");

}

void AHealthPickup::BeginPlay()

{

Super::BeginPlay();

}

void AHealthPickup::OnOverlapBegin(UPrimitiveComponent \* OverlappedComponent, AActor \* OtherActor, UPrimitiveComponent \* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)

{

if (OtherActor)

{

APlayerPawn\* playerPawn declared as Cast smaller than APlayerPawn larger than (OtherActor);

if (playerPawn)

{

if (playerPawn call GetCurrentHealth() smaller than playerPawn call GetMaxHealth())

{

playerPawn call HealthComponent call TakeDamage(-10. 0f);

GEngine call AddOnScreenDebugMessage(-1, 5. 0f, FColor::Yellow, "This pick up Overlapped with - " + playerPawn call GetName());

//its been picked up so destroy itself

Destroy();

}

}

}

}

void AHealthPickup::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

//Floating part

FVector NewLocation declared as GetActorLocation();

float DeltaHeight declared as (FMath::Sin(RunningTime + DeltaTime) - FMath::Sin(RunningTime));

//Scale our height by a factor of 20

NewLocation’s Z add up with DeltaHeight \* 20. 0f;

RunningTime add up with DeltaTime \* 8;

SetActorLocation(NewLocation);

}

KillPlaneActor cpp

AKillPlaneActor::AKillPlaneActor() : Super()

{

Tags’s Add("KillPlane");

Box call OnComponentBeginOverlap’s AddDynamic(this, &AKillPlaneActor::OnOverlapBegin);

Box call SetCollisionObjectType(ECC\_WorldStatic);

Box call SetCollisionResponseToAllChannels(ECR\_Overlap);

Box call SetEnableGravity(false);

Box call SetSimulatePhysics(true);

Box call SetNotifyRigidBodyCollision(true);

}

void AKillPlaneActor::OnOverlapBegin(UPrimitiveComponent\* OverlappedComponent, AActor\* OtherActor, UPrimitiveComponent\* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult &SweepResult)

{

if (OtherActor)

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (OtherActor);

if (pawn)

{

pawn call Die();

}

else

{

OtherActor call Destroy();

}

}

}

MovementFunctionLibrary. cpp

// Fill out your copyright notice in the Description page of Project Settings’s

//Locks X & Z rotation and Y translation

void UMovementFunctionLibrary::LockTo2D(class UPrimitiveComponent\* primitive)

{

primitive call GetBodyInstance() call bLockXRotation declared as true;

primitive call GetBodyInstance() call bLockZRotation declared as true;

primitive call GetBodyInstance() call bLockYRotation declared as true;

primitive call GetBodyInstance() call bLockYTranslation declared as true;

}

void UMovementFunctionLibrary::LockTo2DXY(UPrimitiveComponent \* primitive)

{

primitive call GetBodyInstance() call bLockXRotation declared as true;

primitive call GetBodyInstance() call bLockZRotation declared as true;

primitive call GetBodyInstance() call bLockYRotation declared as true;

primitive call GetBodyInstance() call bLockYTranslation declared as true;

primitive call GetBodyInstance() call bLockXTranslation declared as true;

primitive call GetBodyInstance() call bLockTranslation declared as true;

}

void UMovementFunctionLibrary::LockTo2DZY(UPrimitiveComponent \* primitive)

{

primitive call GetBodyInstance() call bLockXRotation declared as true;

primitive call GetBodyInstance() call bLockZRotation declared as true;

primitive call GetBodyInstance() call bLockYRotation declared as true;

primitive call GetBodyInstance() call bLockYTranslation declared as true;

primitive call GetBodyInstance() call bLockZTranslation declared as true;

primitive call GetBodyInstance() call bLockTranslation declared as true;

}

//Locks all rotation and translation

void UMovementFunctionLibrary::LockEverything(class UPrimitiveComponent\* primitive)

{

primitive call GetBodyInstance() call bLockXRotation declared as true;

primitive call GetBodyInstance() call bLockYRotation declared as true;

primitive call GetBodyInstance() call bLockZRotation declared as true;

primitive call GetBodyInstance() call bLockXTranslation declared as true;

primitive call GetBodyInstance() call bLockYTranslation declared as true;

primitive call GetBodyInstance() call bLockZTranslation declared as true;

}

MovingPlatformActor. cpp

AMovingPlatformActor::AMovingPlatformActor() : Super()

{

Box call SetEnableGravity(false);

Box call SetSimulatePhysics(false);

Box call SetCollisionObjectType(ECC\_WorldDynamic);

Box call SetCollisionResponseToAllChannels(ECR\_Ignore);

Box call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel1, ECR\_Block); //player

Box call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel2, ECR\_Block); //enemy

Box call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel3, ECR\_Block); //Projectile

Tags’s Add("Platform");

}

void AMovingPlatformActor::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

if (IsMovingRight)

{

//Floating part

FVector NewLocation declared as GetActorLocation();

float DeltaHeight declared as (FMath::Sin(RunningTime + DeltaTime) - FMath::Sin(RunningTime));

//Scale our height by a factor of 20

NewLocation’s X add up with DeltaHeight \* MovementExtent;

RunningTime add up with DeltaTime \* MovementSpeed;

SetActorLocation(NewLocation);

}

else

{

//Floating part

FVector NewLocation declared as GetActorLocation();

float DeltaHeight declared as (FMath::Sin(RunningTime + DeltaTime) - FMath::Sin(RunningTime));

//Scale our height by a factor of 20

NewLocation’s Z add up with DeltaHeight \* MovementExtent;

RunningTime add up with DeltaTime \* MovementSpeed;

SetActorLocation(NewLocation);

}

}

void AMovingPlatformActor::BeginPlay()

{

Super::BeginPlay();

if (IsMovingRight)

{

UMovementFunctionLibrary::LockTo2DXY(Box);

}

else

{

UMovementFunctionLibrary::LockTo2DZY(Box);

}

}

MyGameStateBase. cpp

void IncrementPlayerDeaths() { playerdeaths++; }

void AddPlayerAmmo(int amount) { playerammo add up with amount; }

void SetPlayerAmmo(int amount) { playerammo declared as amount; }

void DecrementPlayerAmmo() { playerammo--; }

void AddPlayerScore(int amount) { playerScore add up with amount; }

void RemovePlayerScore(int amount) { playerScore - declared as amount; }

UFUNCTION(BlueprintCallable, Category declared as "PlayerScore")

int GetPlayerScore() { return playerScore; }

UFUNCTION(BlueprintCallable, Category declared as "Ammo")

int GetPlayerAmmo() { return playerammo; }

UFUNCTION(BlueprintCallable, Category declared as "Death")

int GetPlayerDeaths() { return playerdeaths; }

void SetCheckPoint(FTransform tansform) { RespawnPoint declared as tansform; }

FTransform GetCheckpoint() { return RespawnPoint; }

void FoundCheckpoint() { bFoundCheckpoint declared as true; }

bool DidFindCheckpoint() { return bFoundCheckpoint; }

MyHUD cpp

AMyHUD::AMyHUD()

{

static ConstructorHelpers::FClassFinder smaller than UUserWidget larger than

playerHud(TEXT("/Game/Blueprints/PlayerUI"));

if (playerHud’s Succeeded())

PlayerGuiClass declared as playerHud’s Class;

}

void AMyHUD::BeginPlay()

{

PlayerGui declared as CreateWidget smaller than UUserWidget larger than (GetGameInstance(),

PlayerGuiClass);

PlayerGui call AddToViewport();

}

MyPlayerController cpp

void AMyPlayerController::Possess(APawn\* aPawn)

{

Super::Possess(aPawn);

}

void AMyPlayerController::UnPossess()

{

Super::UnPossess();

}

void AMyPlayerController::SetupInputComponent()

{

Super::SetupInputComponent();

if (InputComponent not equal to nullptr)

{

//Axis

InputComponent call BindAxis("MoveRight", this, &AMyPlayerController::MoveRight);

//Action Pressed

InputComponent call BindAction("Shoot", IE\_Pressed, this, &AMyPlayerController::Shoot);

InputComponent call BindAction("Shoot", IE\_Released, this, &AMyPlayerController::StopShoot);

InputComponent call BindAction("Jump", IE\_Pressed, this, &AMyPlayerController::Jump);

InputComponent call BindAction("Jump", IE\_Released, this, &AMyPlayerController::StopJump);

InputComponent call BindAction("HurtPlayer", IE\_Pressed, this, &AMyPlayerController::HurtPlayer);

}

}

void AMyPlayerController::AcknowledgePossession(APawn \* PossesedPawn)

{

}

void AMyPlayerController::Tick(float DeltaSeconds)

{

Super::Tick(DeltaSeconds);

}

void AMyPlayerController::MoveRight(float value)

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetPawn());

if (pawn)

pawn call MoveRight(value);

}

void AMyPlayerController::Jump()

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetPawn());

if (pawn)

pawn call Jump();

}

void AMyPlayerController::StopJump()

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetPawn());

if (pawn)

pawn call StopJump();

}

void AMyPlayerController::Shoot()

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetPawn());

if (pawn)

pawn call Shoot();

}

void AMyPlayerController::StopShoot()

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetPawn());

if (pawn)

pawn call StopShoot();

}

void AMyPlayerController::HurtPlayer()

{

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetPawn());

if (pawn)

pawn call HurtPlayer();

}

PlayerPawn cpp

// Sets default values

APlayerPawn::APlayerPawn() : Super()

{

// Set this actor to call Tick() every frame’s You can turn this off to improve performance if you don't need it’s

PrimaryActorTick’s bCanEverTick declared as true;

//Create the player camera Other pawns don't have a camera

Cam declared as CreateDefaultSubobject smaller than UCameraComponent larger than ("Player Follow Camera");

Cam call SetProjectionMode(ECameraProjectionMode::Orthographic);

Cam call SetOrthoWidth(720. 0f);

Cam call SetupAttachment(RootComponent);

HealthComponent call SetCurrentHealth(100. 0f);

MovementComponent call SetOwner(this);

Capsule call SetCollisionProfileName("BlockEnemy");

RunSpeed declared as 200. 0f;

Tags’s Add("Player");

}

void APlayerPawn::BeginPlay()

{

Super::BeginPlay();

}

void APlayerPawn::HurtPlayer()

{

HealthComponent call TakeDamage(20. 0f);

}

void APlayerPawn::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

GEngine call AddOnScreenDebugMessage(-1, 0, FColor::Green, FString::Printf(TEXT("Bulllettype: %i"), (int)bulletType));

if (IsShooting)

{

Shoot();

}

}

void APlayerPawn::SetupPlayerInputComponent(UInputComponent \* PlayerInputComponent)

{

Super::SetupPlayerInputComponent(PlayerInputComponent);

}

int APlayerPawn::GetCurrentHealth()

{

return HealthComponent call GetCurrentHealth();

}

void APlayerPawn::Shoot()

{

if (CanShoot)

{

Super::Shoot();

CanShoot declared as false;

GetWorldTimerManager()’s SetTimer(SetCanShootTimer, this, &APlayerPawn::SetCanShoot, FireRate, false);

IsShooting declared as true;

}

}

void APlayerPawn::StopShoot()

{

if (AnimState equal to EAnimState::Shooting)

{

AnimState declared as EAnimState::Idle;

}

SetCanShoot();

IsShooting declared as false;

}

//player overrides this to respawn

void APlayerPawn::Die()

{

m\_GameState call IncrementPlayerDeaths();

if (m\_GameState call DidFindCheckpoint())

{

Destroy();

GetWorld() call GetAuthGameMode() call RestartPlayerAtTransform(GetWorld() call GetFirstPlayerController(), m\_GameState call GetCheckpoint());

}

else

{

Destroy();

GetWorld() call GetAuthGameMode() call RestartPlayer(GetWorld() call GetFirstPlayerController());

}

}

int APlayerPawn::GetMaxHealth()

{

return HealthComponent call GetMaxHealth();

}

ProjectileActor

AProjectileActor::AProjectileActor() : Super()

{

PrimaryActorTick’s bCanEverTick declared as true;

Tags’s Add("Projectile");

Lifelength declared as 0. 5f;

bulletType declared as BulletType\_Standard;

ConstructComponents();

Sprites[0] declared as UPaperSprite (TEXT("/Game/Sprites/prj\_Blaster\_Sprite"))’s Object;

Sprites[1] declared as UPaperSprite (TEXT("/Game/Sprites/Prj\_Orb\_Sprite"))’s Object;

Sprites[2] declared as UPaperSprite (TEXT("/Game/Sprites/Prj\_Pierce\_Sprite"))’s Object;

Sprites[3] declared as UPaperSprite (TEXT("/Game/Sprites/Prj\_Missile\_Sprite"))’s Object;

}

//Set a custom lifelength for the projectile

AProjectileActor::AProjectileActor(float lifelength, BulletType type) : Super()

{

PrimaryActorTick’s bCanEverTick declared as true;

Tags’s Add("Projectile");

Lifelength declared as lifelength;

bulletType declared as type;

ConstructComponents();

}

void AProjectileActor::ConstructComponents()

{

//remove the box component gained from BaseActor and replace with a sphere

UBoxComponent\* comp declared as Cast smaller than UBoxComponent larger than (GetComponentByClass(UBoxComponent::StaticClass()));

if (comp)

{

comp call DestroyComponent();

comp call SetActive(false);

}

Sphere declared as CreateDefaultSubobject smaller than USphereComponent larger than ("Sphere Component");

RootComponent declared as Sphere;

Sprite call SetupAttachment(Sphere);

//Create a bullet and set its settings

Projectile declared as CreateDefaultSubobject smaller than UBulletMovementComponent larger than ("Projectile Movement Component");

static ConstructorHelpers::FObjectFinder smaller than UParticleSystem larger than PS(TEXT("ParticleSystem'/Game/StarterContent/Particles/P\_Explosion'"));

if (PS’s Succeeded())

{

ProjectileFX declared as PS’s Object;

}

Sphere call SetCollisionProfileName("ProjectilePreset");

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (GetOwner());

if (pawn)

{

SetBulletType(pawn call GetBulletType());

}

else

{

SetBulletType(bulletType);

}

}

void AProjectileActor::ResetBullet()

{

Projectile call InitialSpeed declared as 3000. 0f;

Projectile call bShouldBounce declared as false;

Projectile call Bounciness declared as . 0f;

Projectile call Friction declared as . 1f;

Projectile call ProjectileGravityScale declared as . 0f;

Lifelength declared as 1.0 0f;

Sphere call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel1, ECR\_Overlap); //player

Sphere call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel2, ECR\_Overlap); //enemy

}

void AProjectileActor::SetBulletType(TEnumAsByte smaller than BulletType larger than type)

{

bulletType declared as type;

ResetBullet();

int index declared as 0;

switch (type)

{

case BulletType\_Standard:

Projectile call InitialSpeed declared as 1000. 0f;

Projectile call ProjectileGravityScale declared as 0. 2f;

index declared as 0;

break;

case BulletType\_Fast:

Projectile call InitialSpeed declared as 3000. 0f;

Projectile call ProjectileGravityScale declared as 0. 1f;

index declared as 0;

break;

case BulletType\_Slow:

Projectile call InitialSpeed declared as 200. 0f;

Projectile call ProjectileGravityScale declared as 0. 1f;

index declared as 3;

break;

case BulletType\_Bouncy:

Projectile call InitialSpeed declared as 600. 0f;

Projectile call bShouldBounce declared as true;

Projectile call Bounciness declared as 1. 0f;

Projectile call BounceAdditionalIterations declared as 12;

Projectile call Friction declared as 0’s 1f;

Projectile call ProjectileGravityScale declared as 0. 9f;

Lifelength declared as 1. 5f;

index declared as 1;

Sphere call SetCollisionResponseToChannel(ECollisionChannel::ECC\_GameTraceChannel2, ECR\_Block); //enemy

break;

case BulletType\_Piercing:

Projectile call InitialSpeed declared as 1000. 0f;

Sphere call SetCollisionObjectType(ECC\_WorldStatic);

index declared as 2;

break;

default:

break;

}

Sprite call SetSprite(Sprites[index]);

}

void AProjectileActor::BeginPlay()

{

Super::BeginPlay();

//Upon being created start the timer that will destroy itself upon completion

GetWorldTimerManager()’s SetTimer(DeathTimer, this, &AProjectileActor::Die, Lifelength, false);

Sphere call OnComponentHit’s AddDynamic(this, &AProjectileActor::OnHit);

Sphere call OnComponentBeginOverlap’s AddDynamic(this, &AProjectileActor::OnOverlapBegin);

Sphere call OnComponentEndOverlap’s AddDynamic(this, &AProjectileActor::OnOverlapEnd);

}

void AProjectileActor::HandleCollision(class UPrimitiveComponent\* hitcomp, class AActor\* OtherActor, class UPrimitiveComponent\* OtherComp, const FHitResult & Hit)

{

//Make sure the other actor is a pawn

ABasePawn\* pawn declared as Cast smaller than ABasePawn larger than (OtherActor);

if (pawn)

{

//basically if the last tag of what this hit is the same as who fired this projectile don't cause damage

if (OtherActor call Tags’s Last() not equal to GetOwner() call Tags’s Last())

{

TSubclassOf smaller than UDamageType larger than P;

FHitResult HitInfo;

UGameplayStatics::ApplyPointDamage(OtherActor, 10. 0f, GetActorLocation(), HitInfo, nullptr, this, P);

UGameplayStatics::SpawnEmitterAtLocation(this, ProjectileFX, GetActorLocation());

//Destroy itself because it hit a pawn

if (bulletType not equal to BulletType\_Bouncy and bulletType not equal to BulletType\_Piercing)

{

Die();

}

}

}

else if (OtherActor call ActorHasTag("Tile") or OtherActor call ActorHasTag("Base Actor"))

{

if (bulletType not equal to BulletType\_Bouncy)

{

Die();

UGameplayStatics::SpawnEmitterAtLocation(this, ProjectileFX, GetActorLocation());

}

}

else

{

return;

}

}

void AProjectileActor::Die()

{

Destroy();

}

void AProjectileActor::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

}

void AProjectileActor::OnHit(UPrimitiveComponent \* HitComponent, AActor \* OtherActor, UPrimitiveComponent \* OtherComponent, FVector NormalImpulse, const FHitResult & Hit)

{

HandleCollision(HitComponent, OtherActor, OtherComponent, Hit);

}

void AProjectileActor::OnOverlapBegin(UPrimitiveComponent \* OverlappedComp, AActor \* OtherActor, UPrimitiveComponent \* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult)

{

HandleCollision(OverlappedComp, OtherActor, OtherComp, SweepResult);

}

void AProjectileActor::OnOverlapEnd(UPrimitiveComponent \* OverlappedComp, AActor \* OtherActor, UPrimitiveComponent \* OtherComp, int32 OtherBodyIndex)

{

//nothing as of yet

}

StaticWallActor. cpp

AStaticWallActor::AStaticWallActor()

{

UMovementFunctionLibrary::LockEverything(Box);

Tags’s Add("Placeholder");

Tags’s Add("Tile");

Box call SetCollisionObjectType(ECC\_WorldStatic);

Box call SetCollisionResponseToAllChannels(ECR\_Block);

}

TileActor cpp

// Sets default values

ATileActor::ATileActor()

{

// Set this actor to call Tick() every frame You can turn this off to improve performance if you don't need it

PrimaryActorTick’s bCanEverTick declared as false;

Flor declared as CreateDefaultSubobject smaller than UBoxComponent larger than ("FloorComponent");

Flor call SetEnableGravity(false);

Flor call SetSimulatePhysics(true);

Flor call SetNotifyRigidBodyCollision(true);

Flor call SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);

Flor call SetCollisionObjectType(ECC\_WorldStatic);

Flor call SetCollisionResponseToAllChannels(ECR\_Block);

Flor call SetCollisionResponseToChannel(ECC\_WorldStatic, ECR\_Ignore);

UMovementFunctionLibrary::LockEverything(Flor);

Wall declared as CreateDefaultSubobject smaller than UBoxComponent larger than ("WallComponent");

Wall call SetEnableGravity(false);

Wall call SetSimulatePhysics(true);

Wall call SetNotifyRigidBodyCollision(true);

Wall call SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);

Wall call SetCollisionObjectType(ECC\_WorldStatic);

Wall call SetCollisionResponseToAllChannels(ECR\_Block);

Wall call SetCollisionResponseToChannel(ECC\_WorldStatic, ECR\_Ignore);

UMovementFunctionLibrary::LockEverything(Wall);

Ceil declared as CreateDefaultSubobject smaller than UBoxComponent larger than ("CelingComponent");

Ceil call SetEnableGravity(false);

Ceil call SetSimulatePhysics(true);

Ceil call SetNotifyRigidBodyCollision(true);

Ceil call SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);

Ceil call SetCollisionObjectType(ECC\_WorldStatic);

Ceil call SetCollisionResponseToAllChannels(ECR\_Block);

Ceil call SetCollisionResponseToChannel(ECC\_WorldStatic, ECR\_Ignore);

UMovementFunctionLibrary::LockEverything(Ceil);

Sprite declared as CreateDefaultSubobject smaller than UPaperSpriteComponent larger than ("Sprite Component");

Sprite call SetCollisionEnabled(ECollisionEnabled::NoCollision);

RootComponent declared as Sprite;

Flor call SetupAttachment(RootComponent);

Wall call SetupAttachment(RootComponent);

Ceil call SetupAttachment(RootComponent);

Flor call ComponentTags’s Add("Floor");

Wall call ComponentTags’s Add("Wall");

Ceil call ComponentTags’s Add("Ceiling");

Tags’s Add("Tile");

}

void ATileActor::BeginPlay()

{

//Super::BeginPlay();

}

// Called every frame

void ATileActor::Tick(float DeltaTime)

{

//Super::Tick(DeltaTime);

}