HISTORY OF UNREAL ENGINE

- Engine development of Unreal Engine began in 1995
- Unreal (game) released in 1998
- Unreal (engine) mostly written by Tim Sweeney (founder of Epic Games)
  - Include features such as collision handling, lighting, advanced texturing, bundled map editor, scripting language, networking support
UNREAL ENGINE 2 AND 3

- Unreal Engine 2 development began in 1999
  - First game released in 2002
  - Improved rendering, and tools pipeline
  - Additional features included physics, particle systems, cinematic editing systems, character animation systems
- Unreal Engine 3 development started in 2002
  - First games released in 2006
  - Engine added support for programmable shader pipeline, and improved physics, graphics, sound, and tools pipeline
  - Additional features included destructible environments, soft-body physics, crowd simulation, global illumination, and multi-platform build targeting
UNREAL ENGINE 4

- Development began in 2003
  - Primarily written by Tim Sweeney in parallel with development of UE3 by the full development team
  - First game, Daylight, released in 2014
- Engine released in 2014 under a subscription model
  - In 2015 this changed to a pure royalties model
- Intended to simplify the scripting systems of previous engine versions and support better global illumination models
  - Major reworks to networking code before the release of Fortnite (allowing more connections with high bandwidth/largescale rendering)
WHAT DOES THIS MEAN FOR THE ENGINE?

- Networked shooter roots
  - Traces of arena-based shooters visible in the underlying class structures
  - Design philosophy built to support this with extensions/modifications to support other genres
- Graphics and networking are “first class” features
  - Highly optimized in both software and hardware support
  - Well supported pipeline
  - Designed to integrate the most modern research possible
- Professional development supported over hobbyist development
  - Yes, Blueprints is intended to bridge this gap
  - But system fundamentally assumes large teams and expert systems users
WORKING WITH MODULES

- Games, programs, and the UE4 editor itself are all targets built by the UnrealBuildTool
  - Compiled from C++ modules, or areas of functionality
  - Build rules allow modules to interact
- C# scripts determine build rules and included modules
  - These are the .cs files generated within the Source folder
MAJOR MODULE CATEGORIES

- Runtime
  - Features for efficiently creating and running a game
  - Basis of gameplay programming (our primary focus in this class)

- Editor
  - Features for working within the Editor or building out Editor tools
  - Underlying systems that support gameplay programming

- Developer
  - Features related to outside assets and tools that may require interfacing or modification
  - Assists with asset management, testing suites, profiling and other features not within the editor

- Plugins
  - Features useful for runtime, editor, or developer, but are not within these three categories
  - Added as benefits the project
RUNTIME

- **Core** provides common framework for UE4 modules to communicate as well as math and container libraries and hardware support

- **CoreUObject** defines UObject type allowing for reflection, garbage collection, and serialization within the runtime system

- **Engine** contains game functionality and types that support it, such as Actors, Components, and Gameplay

- Other modules supported include advertising, analytics, AR/VR, networking, physics, rendering, AI, GUI, audio, file parsing, etc...
**EDITOR**

- **Kismet** provides Blueprint editor functionality and is supported by **KismetCompiler** and **KismetWidget**

- **LevelEditor** contains level editing functionality and viewing tools

- **PropertyEditor** contains functionality for displaying and editing UProperties

- Other modules include support for landscape painting, mesh editing, animations, AI, inputs, level streaming, light building, and basically anything else that involves the Editor
DEVELOPER

- **AutomationController** and **AutomationWindow** used to connect to automation system

- **OutputLog**, **GameplayDebugger**, and **Profiler** (among many others) provide debug information and profiling tools

- **DeviceManager** provides interface for interacting with connected devices

- Other modules include support for mesh and texture handling, build systems, deployment, audio tools and anything else related to the tools pipeline and not the editor or gameplay directly
PLUGINS

- **Paper2D, Paper2DEditor, PaperSpriteSheetImporter**, and **PaperTiledImporter** provide sprite and flip-book (e.g. sprite animation) support as well as sprite-based collision and sprite importing.

- **PhysXVehicles** and **PhysXVehiclesEditor** provide support for creating vehicle physics.

- **SteamVR** and **SteamVREditor** provide support for Steam VR services.

- Other modules include any potentially useful, but specialized, functionality related to gameplay, editor or developer categories.
RUNTIME MODULES

- Main focus of this class!

- Other categories are incredibly important but game engines are just too vast to explore in a single semester

- Gameplay programming is likely the most familiar, accessible aspect of all this
**The GameModeBase defines the game being played. It governs the game rules, scoring, what actors are allowed to exist in this game type, and who may enter the game.**

*It is only instanced on the server and will never exist on the client.*

*A GameModeBase actor is instantiated when the level is initialized for gameplay in C++ UGameEngine::LoadMap().*

*The class of this GameMode actor is determined by (in order) either the URL ?game=xxx, the GameMode Override value set in the World Settings, or the DefaultGameMode entry set in the game’s Project Settings.*

* @see [https://docs.unrealengine.com/latest/INT/Gameplay/Framework/GameMode/index.html](https://docs.unrealengine.com/latest/INT/Gameplay/Framework/GameMode/index.html)*

```cpp
UCLASS(config = Game, notplaceable, BlueprintType, Blueprintable, Transient, hideCategories = (Info, Rendering, MovementReplication, Replication, Actor), meta = (ShortTooltip = "Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type."))
class ENGINE_API AGameModeBase : public AInfo
{

GENERATED_UCLASS_BODY()
```

*AInfo is a Actor base class that does not need physical representation in the world (e.g. a manager)
GAMEMODEBASE CONSTRUCTOR

AGameModeBase::AGameModeBase(const FObjectInitializer& ObjectInitializer)
: Super(ObjectInitializer.DoNotCreateDefaultSubobject(TEXT("Sprite")))
{
    bNetLoadOnClient = false;
    bPauseable = true;
    bStartPlayersAsSpectators = false;

    DefaultPawnClass = ADefaultPawn::StaticClass();
    PlayerControllerClass = APlayerController::StaticClass();
    PlayerStateClass = APlayerState::StaticClass();
    GameStateClass = AGameStateBase::StaticClass();
    HUDClass = AHUD::StaticClass();
    GameSessionClass = AGameSession::StaticClass();
    SpectatorClass = ASpectatorPawn::StaticClass();
    ReplaySpectatorPlayerControllerClass =
        APlayerController::StaticClass();
    ServerStatReplicatorClass = AServerStatReplicator::StaticClass();
}

*AInfo has a sprite component for displaying in Editor that we do not want to create*
GAMEMODEBASE RESETLEVEL

/**
 * Overridable function called when resetting level. This is used to reset the game state while
 * staying in the same map
 * Default implementation calls Reset() on all actors except GameMode and Controllers
 */

UFUNCTION(BlueprintCallable, Category=Game)
virtual void ResetLevel();
void AGameModeBase::ResetLevel() {
    UE_LOG(LogGameMode, Verbose, TEXT("Reset %s"), *GetName());

    // Reset ALL controllers first
    for (FConstControllerIterator Iterator = GetWorld()->GetControllerIterator();
        Iterator; ++Iterator) {
        AController* Controller = Iterator->Get();
        APlayerController* PlayerController = Cast<APlayerController>(Controller);
        if (PlayerController) {
            PlayerController->ClientReset();
        }
        Controller->Reset();
    }

    // Reset all actors (except controllers, the GameMode, and any other actors specified by ShouldReset())
    for (FActorIterator It(GetWorld()); It; ++It) {
        AActor* A = *It;
        if (A && !A->IsPendingKill() && A != this && !A->IsA<AController>() && ShouldReset(A)) {
            A->Reset();
        }
    }

    // Reset the GameMode
    Reset();

    // Notify the level script that the level has been reset
    ALevelScriptActor* LevelScript = GetWorld()->GetLevelScriptActor();
    if (LevelScript) {
        LevelScript->LevelReset();
    }
}
Return the 'best' player start for this player to spawn from
Default implementation looks for a random unoccupied spot
@param Player is the controller for whom we are choosing a playerstart
@return AActor chosen as player start (usually a PlayerStart)
*/

UFUNCTION(BlueprintNativeEvent, Category=Game)
AActor* ChoosePlayerStart(AController* Player);
AActor* AGameModeBase::ChoosePlayerStart_Implementation(AController* Player) {

    // Choose a player start
    APlayerStart* FoundPlayerStart = nullptr;
    UClass* PawnClass = GetDefaultPawnClassForController(Player);
    APawn* PawnToFit = PawnClass ? PawnClass->GetDefaultObject<APawn>() : nullptr;
    TArray<APlayerStart*> UnOccupiedStartPoints;
    TArray<APlayerStart*> OccupiedStartPoints;
    UWorld* World = GetWorld();
    for (TActorIterator<APlayerStart> It(World); It; ++It) {
        APlayerStart* PlayerStart = *It;

        if (PlayerStart->IsA<APlayerStartPIE>()) {
            // Always prefer the first "Play from Here" PlayerStart, if we find one while in PIE mode
            FoundPlayerStart = PlayerStart;
            break;
        } else {
            FVector ActorLocation = PlayerStart->GetActorLocation();
            const FRotator ActorRotation = PlayerStart->GetActorRotation();
            if (!World->EncroachingBlockingGeometry(PawnToFit, ActorLocation, ActorRotation)) {
                UnOccupiedStartPoints.Add(PlayerStart);
            } else if (World->FindTeleportSpot(PawnToFit, ActorLocation, ActorRotation)) {
                OccupiedStartPoints.Add(PlayerStart);
            }
        }
    }

    if (FoundPlayerStart == nullptr) {
        if (UnOccupiedStartPoints.Num() > 0) {
            FoundPlayerStart = UnOccupiedStartPoints[FMath::RandRange(0, UnOccupiedStartPoints.Num() - 1)];
        } else if (OccupiedStartPoints.Num() > 0) {
            FoundPlayerStart = OccupiedStartPoints[FMath::RandRange(0, OccupiedStartPoints.Num() - 1)];
        }
    }

    return FoundPlayerStart;
}
// Delegate signatures
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_FiveParams( FTakeAnyDamageSignature, 
AActor, OnTakeAnyDamage, AActor*, DamagedActor, float, Damage, const class 
UDamageType*, DamageType, class AController*, InstigatedBy, AActor*, 
DamageCauser );
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_NineParams( FTakePointDamageSignature, 
AActor, OnTakePointDamage, AActor*, DamagedActor, float, Damage, class 
AController*, InstigatedBy, FVector, HitLocation, class UPrimitiveComponent*, 
FHitComponent, FName, BoneName, FVector, ShotFromDirection, const class 
UDamageType*, DamageType, AActor*, DamageCauser );
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_SevenParams( FTakeRadialDamageSignature, 
AActor, OnTakeRadialDamage, AActor*, DamagedActor, float, Damage, const class 
UDamageType*, DamageType, FVector, Origin, FHitResult, HitInfo, class AController*, 
InstigatedBy, AActor*, DamageCauser );
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams( FActorBeginOverlapSignature, 
AActor, OnActorBeginOverlap, AActor*, OverlappedActor, AActor*, OtherActor );
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams( FActorEndOverlapSignature, 
AActor, OnActorEndOverlap, AActor*, OverlappedActor, AActor*, OtherActor );
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_FourParams( FActorHitSignature, AActor, 
OnActorHit, AActor*, SelfActor, AActor*, OtherActor, FVector, NormalImpulse, const 
FHitResult&, Hit );

*Sparse delegates are delegates that are infrequently bound*
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_OneParam(FActorBeginCursorOverSignature, AActor, OnBeginCursorOver, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_OneParam(FActorEndCursorOverSignature, AActor, OnEndCursorOver, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorOnClickedSignature, AActor, OnClicked, AActor*, TouchedActor, FKey, ButtonPressed);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorOnReleasedSignature, AActor, OnReleased, AActor*, TouchedActor, FKey, ButtonReleased);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorOnInputTouchBeginSignature, AActor, OnInputTouchBegin, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorOnInputTouchEndSignature, AActor, OnInputTouchEnd, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorBeginTouchOverSignature, AActor, OnInputTouchEnter, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorEndTouchOverSignature, AActor, OnInputTouchLeave, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_OneParam(FActorDestroyedSignature, AActor, OnDestroyed, AActor*, DestroyedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorEndPlaySignature, AActor, OnEndPlay, AActor*, Actor, EEndPlayReason::Type, EndPlayReason);

DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorOnInputTouchBeginSignature, AActor, OnInputTouchBegin, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorOnInputTouchEndSignature, AActor, OnInputTouchEnd, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorBeginTouchOverSignature, AActor, OnInputTouchEnter, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorEndTouchOverSignature, AActor, OnInputTouchLeave, ETouchIndex::Type, FingerIndex, AActor*, TouchedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_OneParam(FActorDestroyedSignature, AActor, OnDestroyed, AActor*, DestroyedActor);
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_TwoParams(FActorEndPlaySignature, AActor, OnEndPlay, AActor*, Actor, EEndPlayReason::Type, EndPlayReason);

UCLASS(BlueprintType, Blueprintable, config=Engine, meta=(ShortTooltip="An Actor is an object that can be placed or spawned in the world."))
class ENGINE_API AActor : public UObject
{
    GENERATED_BODY()
}
void AActor::InitializeDefaults() {
    PrimaryActorTick.TickGroup = TG_PrePhysics;
    // Default to no tick function, but if we set 'never ticks' to false (so there is a tick function) it is enabled by default
    PrimaryActorTick.bCanEverTick = false;
    PrimaryActorTick.bStartWithTickEnabled = true;
    PrimaryActorTick.SetTickFunctionEnable(false);

    CustomTimeDilation = 1.0f;

    SetRole(ROLE_Authority);
    RemoteRole = ROLE_None;
    bReplicates = false;
    NetPriority = 1.0f;
    NetUpdateFrequency = 100.0f;
    MinNetUpdateFrequency = 2.0f;
    bNetLoadOnClient = true;
#if WITH_EDITORONLY_DATA
    bEditable = true;
bListedInSceneOutliner = true;
bIsEditorPreviewActor = false;
bHiddenEdLayer = false;
bHiddenEdTemporary = false;
bHiddenEdLevel = false;
bActorLabelEditable = true;
    SpriteScale = 1.0f;
bEnableAutoLODGeneration = true;
bOptimizeBPComponentData = false;
#endif // WITH_EDITORONLY_DATA

*Called by all constructors
NetCullDistanceSquared = 225000000.0f;
NetDriverName = NAME_GameNetDriver;
NetDormancy = DORM_Awake;
// will be updated in PostInitProperties
bActorEnableCollision = true;
bActorSeamlessTraveled = false;
bBlockInput = false;
SetCanBeDamaged(true);
bFindCameraComponentWhenViewTarget = true;
bAllowReceiveTickEventOnDedicatedServer = true;
bRelevantForNetworkReplays = true;
bRelevantForLevelBounds = true;

// Overlap collision settings
bGenerateOverlapEventsDuringLevelStreaming = false;
UpdateOverlapsMethodDuringLevelStreaming = EActorUpdateOverlapsMethod::UseConfigDefault;
DefaultUpdateOverlapsMethodDuringLevelStreaming = EActorUpdateOverlapsMethod::OnlyUpdateMovable;

bHasDeferredComponentRegistration = false;
#if WITH_EDITORONLY_DATA
PivotOffset = FVector::ZeroVector;
#endif
SpawnCollisionHandlingMethod = ESpawnActorCollisionHandlingMethod::AlwaysSpawn;

#if (CSV_PROFILER && !UE_BUILD_SHIPPING)
// Increment actor class count
{
    if (!HasAnyFlags(RF_ArchetypeObject | RF_ClassDefaultObject)) {
        FScopeLock Lock(&CSVActorClassNameToCountMapLock);

        const UClass* ParentNativeClass = GetParentNativeClass(GetClass());
        FName NativeClassName = ParentNativeClass ? ParentNativeClass->GetFName() : NAME_None;
        int32& CurrentCount = CSVActorClassNameToCountMap.FindOrAdd(NativeClassName);
        CurrentCount++;
        CSVActorTotalCount++;
    }
}
#endif // (CSV_PROFILER && !UE_BUILD_SHIPPING)
OBSERVATIONS

- UE4 classes are quite complex and file structure is difficult to navigate without more advanced search features in an IDE.
- Code itself is designed to be highly readable.
  - Verbose naming
  - Spare but clear in-line comments
- Relatively easy to explore if you need to understand some functionality more deeply.
  - Learn the systems as you encounter the systems.
TAKE AWAYS

- Advanced software systems (like game engines) are extremely large and complex
  - Understanding the use cases of a system make it more accessible
  - Patience and persistence is essential
  - Progress early on will be slow and steady
  - Try to solve issues on your own but don’t be afraid to ask for help
FURTHER READING

- Full API of all UE4 modules <https://docs.unrealengine.com/en-US/API/index.html>