Unreal Engine AI and Game Code Overview

Steven Polge
Epic Games
October 16, 2001

PART I - Engine Classes defined in UnrealScript

Actor

Base class of all gameplay objects.

A large number of properties, behaviors and interfaces are implemented in the base Actor class, including:

- Display
- Animation
- Physics and world interaction (discussed later)
- Making sounds
- Networking properties (discussed in networking session).
- Actor creation and destruction (discussed later)
- Triggering and timers
  - When an actor with a defined event calls TriggerEvent() or UntriggerEvent(), the Trigger() or Untrigger() function is called for all actors with the matching tag.
  - An actor's event is triggered for the following events:
    - Decoration - when it's destroyed.
    - PlayerStart - when a pawn spawns in it.
    - GameInfo - when the game ends, it triggers an event named 'EndGame'.
    - Mover - when it finisheds opening.
    - Mover - when it bumps an actor, it triggers its BumpEvent or PlayerBumpEvent.
    - Pawn - when it's killed.
    - Pickup - when it's picked up.
    - Teleporter - when it's used.
    - Trigger - when touched by a relevant actor, or damaged if it's a damageable trigger.
    - PhysicsVolume - when a player enters the volume.
- Actor iterator functions
  - AllActors is slooow.
  - DynamicActors faster, since it skips all actors with bStatic==true
  - TouchingActors very fast (goes through actors Touching array)
  - CollidingActors fast for relatively small radii (uses collision hash)
    - Radius relative to level size - as long as only a small percentage of actors will be considered
- Message broadcasting

Tick() and PlayerTick() - called every frame
- PlayerControllers get PlayerTick()
- Avoid implementing tick() - scripts should be event driven to be efficient

Pawn and Controller

Pawn is physical representation of players/ NPC AIs in a level.
- Pawn specific physics properties (movement speed, etc.)
- AI related flags (hearing, seeing capabilities
- Weapon/Inventory handling functions - adding, removing, finding, selecting
- TakeDamage(), Dying state
- *NEW* animation interface for pawns
  - Animations are generated client-side, reducing network bandwidth
  - Pawns handle animend(), not their controller (except for ScriptedControllers)
  - Per tick blending changes for smooth movement transitions
  - Still prototype - will use animation object interface

Controllers are non-physical actors which can be attached to a pawn to control its actions.
- **PlayerControllers** are used by human players to control pawns
- **AIControllers** are used to implement the artificial intelligence for the pawns they control.
  - **AIScripts** can be associated with pawns placed in levels to modify their AIControllers.
- **ScriptedControllers** can be used to make a pawn follow a scripted sequence, defined by **ScriptedSequence** (a subclass of AIScript) actors.
- Controllers use Possess() and UnPossess() to take or relinquish control of a pawn.
- Controllers receive notifications for many of the events occurring for the pawn they are controlling, giving them the opportunity to intercept the event and supercede the Pawn's default behavior.

PlayerController
- Player control pre-processed by PlayerInput object
- PlayerTick() called every frame to allow PlayerController to control pawn based on player inputs.
- Player Movement states (PlayerWalking, PlayerSwimming, etc) for each mode that has different control.

GameInfo and related classes

Defines the game being played: the game rules, scoring, what actors are allowed to exist in this game type, and who may enter the game.

GameInfo actor class is determined by (in order) either the DefaultGameType if specified in the LevelInfo, or the DefaultGame entry in the game's .ini file (in the Engine.Engine section), unless it's a network game in which case the DefaultServerGame entry is used

The GameInfo's InitGame() function is called before any other scripts (including PreBeginPlay()), and is used by the GameInfo to initialize parameters and spawn its helper classes.

The login process

- Used even in single player game
- In a network game**,** the **AccessControl** class determines whether or not player is allowed to login in PreLogin() function. It also controls whether a player can enter as a participant, a spectator, or a game administrator.
- Ulevel::SpawnPlayActor() calls the GameInfo Login() function to spawn a player controller, then attaches a Player to the returned PlayerController, then handles traveling inventory and properties.
- The GameInfo Login() function
  - Sets the player team, if relevant (by calling PickTeam() and ChangeTeam())
  - Finds an appropriate player start (by calling FindPlayerStart())
  - Initializes the PlayerReplicationInfo
  - Validates the desired pawn class.
  - If not a delayed start, start match or spawn player pawn immediately, else wait for match to start.

Mutators allow modifications to gameplay while keeping game rules intact. Multiple mutators can be used together. (intended for mod authors)

- ModifyLogin() used to modify player login parameters.
- ModifyPlayer() used to modify player pawn properties.
- GetDefaultWeapon() used to modify the default weapon for players.
- CheckRelevance() used to modify, replace, or remove all actors. Called from the PreBeginPlay() function of all actors except those (Decals, Effects and Projectiles for performance reasons) which have bGameRelevant==true.

GameRules specify optional modifications to game rules, such as scoring, finding player starts, and damage modification. (intended for mod authors)

BroadcastHandler handles both text messages (typed by a player) and localized messages (which are identified by a LocalMessage class and id).

- GameInfos produce localized messages using their DeathMessageClass and GameMessageClass classes.

PlayerReplicationInfo, GameReplicationInfo and TeamInfo

Are always relevant, and contain replicated attributes which are important to keep

updated for all clients.
Each player has an associated PlayerReplicationInfo, and the GameInfo has an associated GameReplicationInfo.

LevelInfo

Each level has one LevelInfo, automatically generated by the level editor when the level is created.
Always the first actor in the actor list.
Holds properties of global importance in the level, such as the time (TimeSeconds), and the networking mode (NetMode - server or client).
All actors in the level have access to the LevelInfo through their Level attribute.
Has a reference to the level's GameInfo actor through its Game attribute. While there is a valid LevelInfo actor for all servers and clients, only servers and standalone games have GameInfos.
The LevelInfo also contains two specialized actor lists that are used for fast access to certain actor types. The ControllerList is a linked list of all controllers in the level, and the NavigationPointList is a linked list of all NavigationPoints in the level. These lists will probably become obsolete when we change the Actor list to a TMap.

Volume, PhysicsVolume, and BlockingVolume

Used for defining areas with gameplay implications.
Touch() and Untouch() notifications to the volume as actors enter or leave it
ActorEnteredVolume() and ActorLeftVolume() notifications when center of actor enters the volume
Pawns with bIsPlayer==true cause PlayerEnteredVolume() and PlayerLeftVolume() notifications instead.
AssociatedActor also gets touch() and untouch() notifications (for example, to create non-cylindrical triggers).
BlockingVolumes used to provide fast, simple collision (around static meshes for example). By default, they collide with non-zero extent traces only.
PhysicsVolumes contain properties which affect physics of actors in them (gravity, etc.)

- This functionality used to be in ZoneInfo.
- Priority attribute determines which PhysicsVolume has precedence.
- PhysicsVolumes also have built in support for entry and exit sounds/actors.
- PhysicsVolumes can cause recurring damage to actors in them.

NavigationPoints

Organized into network to provide AIControllers the capability of determining paths to arbitrary destinations in a level.
Each NavigationPoint has a PathList of ReachSpecs which describe paths which can be reached from that node.
Each ReachSpec specifies a destination, and the movement requirements (size, physics modes, etc.) required to take that path.
NEW UpstreamPaths[] and PrunedPaths[] were removed from the latest code, and Reachspecs are now UnrealScript defined actors. PathList is now a dynamic array.
Special NavigationPoint types (door, ladder, liftcenter and liftexit) used to specify navigation in conjunction with movers. Interface for telling AI how to use these paths described later.

Inventory, Pickup, and AttachedInventory

Pickup is the base class of actors that when touched by an appropriate pawn, will create and place an Inventory actor in that pawn's inventory chain.

Has an associated inventory class (its InventoryType).
Placed by level designers.
Can only interact with pawns when in their default Pickup state. Pickups verify that they can give inventory to a pawn by calling the GameInfo's PickupQuery() function. After a pickup spawns an inventory item for a pawn, it then queries the GameInfo by calling the GameInfo's ShouldRespawn() function about whether it should remain active, enter its Sleep state and later become active again, or destroy itself.
Has an AI interface to allow AIControllers, such as bots, to assess the desireability of acquiring that pickup. The BotDesireability() method returns a float typically between 0 and 1 describing how valuable the pickup is to the AIController. This method is called when an AIController uses the FindPathToBestInventory() navigation intrinsic.
When navigation paths are built, each pickup has an InventorySpot (a subclass of NavigationPoint) placed on it and associated with it (the Pickup's MyMarker== the InventorySpot, and the InventorySpot's markedItem == the pickup).

Inventory is the parent class of all actors that can be carried by other actors.

Placed in the holding actor's inventory chain, a linked list of inventory actors.
Each inventory class knows what pickup can spawn it (its PickupClass).
When tossed out (using the DropFrom() function), inventory items replace themselves with an actor of their Pickup class.
Most Inventory actors are never rendered. The common exception is Weapon actors. Inventory actors may be rendered in the first person view

of the player holding them, with the Inventory function, using the RenderOverlays() function. The CalcDrawOffset() function determines where to render the item on the player's screen.
Inventory items may also be rendered attached to the player's mesh, by spawning an appropriate **InventoryAttachment** actor.

Weapon, AttachedWeapon, Projectile and Ammunition

Pawns use weapons by calling the weapon's fire() or altfire() function. Each pawn has one currently active weapon (specified by its Weapon attribute).
NEW All weapons require ammunition. When a weapon is given to a pawn, it will spawn the appropriate **Ammunition** actor (as determined by the weapon's AmmoName attribute) in the pawn's inventory chain if it does not exist, or adding the weapon's PickupAmmoCount to the ammunition if it does. Whenever a weapon fires, it will first call its Ammunition's UseAmmo() function to verify that ammunition is available, and if so reduce the remaining ammunition.
NEW Ammunition is now responsible for spawning the appropriate **Projectile** or processing a trace hit. This allows weapons to have multiple ammunition types each with different behavior (replaces old fire()/altfire() behavior).
NEW weapon firing code updated - described in detail in networking session.
Weapon AI interface used for picking the appropriate weapon (the RecommendWeapon() function, which compares the value of the weapons available in the inventory chain), and determining the tactics to use with it. RateSelf() specifies how valuable the weapon is in the controller's current tactical situation. SuggestAttackStyle() tells the controller whether it should be aggressive or cautious when using this type of weapon, while SuggestDefenseStyle() tells the controller whether it should be aggressive or cautious when being attacked by an enemy wielding this weapon. Ammunition now also has an AI interface (RateSelf()) because of its expanded role.

Mover

Movers are Actors with a StaticMesh that moves between its keyframes when triggered depending on its initial state.
Movers send notifications to AIs that have sent them as their PendingMover.

Trigger

When enabled generates events when triggered by an appropriate actor (usually by touching, or by shooting.

Effects

Base class of all gratuitous special effects.
Generally should not be replicated, but rather spawned on client side by other replicated actors.

Damagetype

Abstract classes which are responsible for specifiying many damage related attributes, such as the effects (blood, screen flash, etc.) associated with that damage, and the string to print to describe deaths by that type of damage.
Passed as a parameter of the actor TakeDamage() function.

LocalMessage

Abstract classes which contain an array of localized text .
The PlayerController function ReceiveLocalizedMessage() is used to send messages to a specific player by specifying the LocalMessage class and index. This allows the message to be localized on the client side, and saves network bandwidth since the text is not sent.

HUD and Scoreboard

The **HUD** is responsible for drawing any information overlay.

- The local player always has a valid HUD.
- The HUD type is defined by the GameInfo actor.
- Every frame, the HUD's postrender function is called after the world has been rendered.
- ShowDebug exec will show debug parameters of currently viewed actor (use ViewClass xxx to change viewed actor).

PART II - Game code in C++

Navigation AI

Reachspecs must be built before navigation network can be used (in the UnrealEd build menu).
Controllers can check if a nearby point or actor (less than MAXPATHDIST away) is directly reachable using PointReachable() and ActorReachable().
MoveToward() and MoveTo() are latent functions which cause the Controller's pawn to move toward the specified destination. State code execution continues when either the destination is reached, or progress is no longer possible.

- If the move is from one NavigationPoint to another, the destination NavigationPoint's SuggestMovePreparation() is called if it is implemented, to allow it to direct the pawn to perform some action first.

The AIController functions WaitForMover() and MoverFinished() provide an interface between a mover/its navigationpoint and the controller.
If the current pawn's collision or other properties are not supported by the path between the navigation points, the AIController's PrepareForMove() is called (to allow it to crouch, for example).
While the pawn's bPreparingMove== true, the movement is suspended.

For destinations that aren't directly reachable, FindPathToward() and FindPathTo () will return the NavigationPoint to move directly toward to reach that destination. When the NavigationPoint is reached, call FindPathxxx() again to determine the next path (with any dynamic path network changes considered).

The Controller's RouteCache[] array contains the first 16 NavigationPoints in the best path determined toward the destination.

NodeEvaluator functions can be defined to specify node desireability for routing when a specific destination is not specified. Path finding code drops out immediately if result is >= 1.0. FindPathTowardNearest(), FindRandomDest() are example native script functions which take advantage of this capability.

Actor creation and destruction

Spawn() in script calls SpawnActor() in C++

For script spawned actors, the spawned actor's instigator is automatically set to be the instigator of the actor which is calling Spawn().
Actor must fit where it is placed in the world, or it won't be spawned.
After initialization (with exceptions noted below), the following actor events are called:

Spawned() - C++
PreBeginPlay() - script (handles destruction if not game relevant)
BeginPlay() - script
PostBeginPlay() - script
PostNetBeginPlay() - script (Called after replicated properties of actor have been updated - note that replication at this point isn't guaranteed)
If actor has an auto state, its BeginState() is called.

Final initialization of the actor is done just before calling PostBeginPlay() in the following order:

The actor's ZoneInfo and PhysicsVolume are set.
Collision with blocking non-world geometry actors is resolved using actor events EncroachingOn() and EncroachedBy().

Touching notifies currently don't happen when actor is spawned.

Actor creation at level startup

InitGame is called on the GameInfo
Spawned() is not called.
PreBeginPlay() is called for all actors
BeginPlay() called for all actors
ZoneInfos and Volumes set for all actors

PostBeginPlay() called for all actors
PostNetBeginPlay() called for all actors
BeginState() is called for all actors with initial state

Actor destruction

Initially bDeleteMe is set
When 255 actors marked for deletion, they are cleaned up, with all actor references to other actors removed.
Currently, deleted actors may still be visible to script before cleanup. Will be changed in next version - note potential issues to avoid.

Physics and world interaction

Touch() and Untouch() notifications used for collisions between actors for whom collision is enabled, but which don't block each other.

Occurs during one actor's physics. Avoid infinite loops (singular keyword is simple fix).
Use PendingTouch actor list for actors which want to add an effect after the move completes using the PostTouch() notification.

Bump() notification sent when actors which block each other collide.

Occurs during one actor's physics. Avoid infinite loops (singular keyword is simple fix).

Base/Attached actors

When actor gets its base set, it is added to the base's Attached array
Base gets Detach() and Attach()notifications
Actor gets BaseChanged() notification
Base can change when:
Change physics mode
Teleport (lose attached actors)
Pawn walking
SetBase() from script
Special case if AttachmentBone!=None

Actor Physics modes

PHYS_Projectile
PHYS_Falling:
PHYS_Rotating: Rotation, no translation
PHYS_Trailer: soon obsolete
PHYS_RootMotion: under construction
Rotation
Not updated if PHYS_None
If bFixedRotationDir==true, will continue rotating in same direction, even after reaching DesiredRotation.
If bRotateToDesired==true, will rotate to DesiredRotation and stop.
Physics notifications:
HitWall()

- Landed() (HitNormal.Z > MINFLOORZ)

Pawn physics modes

- PHYS_Walking
  - Optional Check for ledges - MayFall() notification
    - For AI if !bCanWalkOffLedges, with optional bAvoidLedges (keep away from them).
  - MinHitWall to limit HitWall() notifications
- PHYS_Falling
- PHYS_Flying
- PHYS_Swimming
- PHYS_Spider
- PHYS_Ladder
- PHYS_RootMotion: under construction
- Rotation (APawn::PhysicsRotation())
  - bCrawler to orient in floor direction
  - No pitching when on ground.
  - If not bCrawler, roll when angular momentum
- Crouching:
  - Can only crouch if bCanCrouch==true
  - To request crouch, set bWantsToCrouch
  - bIsCrouched==true while crouched
  - bTryToUncrouch is true for AI pawns which automatically crouched during movement - they continually try to stand up.