Lumberyard

Developer Guide

Version 1.1

Lumberyard: Developer Guide

Table of Contents

Lumberyard for Programmers

The Lumberyard Developer Guide is intended for programmers or anyone working directly with the Lumberyard code.

This guide includes the following sections:

AI

This section describes the AI system. It includes a general overview of key concepts, describes system components, and provides an AI scripting manual.

This section includes the following topics:

AI System Concepts

Key features of the AI system include the following:

Navigation

- Navigation with little or no assistance from the designers
- Multi-layer navigation (flying, swimming, zero-gravity) or simple 2D navigation
- Smart objects for special navigation and interactions

Individual AI

- Easy-to-use static and dynamic covers (such as behind movable vehicles)
- Dynamic tactical points (such as cover points, ambush points, patrol waypoints)
- Behavior trees, to select behaviors based on values of Boolean variables
- Customizable perception (such as vision, sound, memory, sixth sense)

Group and Global AI

- Group behavior trees, to define group tactics

- Formations, to move AI characters in some orderly fashion
- Factions (such as friends, neutrals, enemies)
- Visual flow graphs of game logic, with macro-nodes for reused sub-flow graphs

MMO-ready

- Support for streaming big maps

User-friendly

- Visual AI debugger to log signals, behavior changes, goal changes, user comments
- Behavior tree visual editor
- Visual flow graph editor and debugger (with visual flow propagation and break points)

This section includes the following topics:

AI System Overview

This section outlines basic concepts related to the AI system.

Navigation

- Default navigation system
  - Triangulation
    - 2D terrain-based navigation
    - Uses cylindrical objects (such as trees) and forbidden areas
  - Navigation modifiers
    - Human waypoints – Need to be place manually but connections can be generated automatically
    - Flight – Information about navigable volumes for flying entities
    - Volume – General volume navigation, such as for oceans
- Multi-layer navigation system
- Smart object system: allows AI agents to move in special ways
- AI territories & waves
  - Control number of active AI agents (through flow graph logic)
  - Activate, deactivate, and spawn all AI agents assigned to a territory using a single FG node
  - AI waves can be attached to AI territories and allow independent AI activations
  - AI waves automatically handle entity pool issues for assigned AI agents, such as loading/unloading

In general, a search is time-sliced to use 0.5 ms per AI frame (configured using the console variable `ai_PathfinderUpdateTime`). Options for pathfinding techniques include high priority, straight, and partial. Updates for human waypoints are heavy but time-sliced. The navigation graph is optimized but needs memory. Navigation data is generated offline in Editor. With multi-layer navigation, the navigation mesh is regenerated when the designer modifies the map.

Decision Making

- Behavior selection system – Uses behavior trees to select AI behaviors
- Cover system – Provides AI agents with static and dynamic covers
- Smart object system – Allows AI agents to interact with their environment
- Interest system – Allows AI agents to perform intelligent actions when not alerted

Tactical

- Tactical point system (TPS) – Allows AI agents to ask intelligent questions about their environment (such as where to hide or where to attack)
- Faction system – Determines levels of hostility between AI agents
- Group coordination system – Uses coordination selection trees to select group behaviors
- Formation system – Allows AI agents to move in formations
- Cluster detector – detects clusters of points in space and subdivides them into separate groupings that satisfy specific properties (using a modified K-mean algorithm); used with AISquadManager to group different AI agents into dynamic squads

World-Interfacing

- Signals – To trigger events and/or change behaviors
- Perception system
  - Perception handler (legacy, usually per game)
  - Target track system – Uses configurable ADSR envelopes to represent incoming stimuli
- Communication system – Allows AI agents to play sound/voice/animation events

Development Environment

The design and development environment includes the following components:

- Game object model – Entity, movement controller, extensions
- Actor & vehicle system – Health, camera, IK, weapons, animation, etc.
- Flow graph – Visual definition of game logic
- AI debug renderer – HUD, geometric primitives, text labels, graphs, etc.
- Editor
  - AI entities – Properties, flow graphs, scripts
  - Entity archetypes – Templates for properties of individual AI agents
  - AI shapes – AI territories, AI paths, forbidden areas
  - Navigation – Navigation modifiers used instead of triangulation
  - Cover surfaces – CoverSurface anchors to indicate where cover should be
  - Visual AI debugger – Recording AI signals, active behaviors, goals, stimuli, etc.
- Scripting with Lua
  - Entity definitions (including entity flow graph nodes)
  - AI behavior definitions
  - Group behavior definitions

  - Library or shared Lua code (game rules, basic entities)
  - Blackboards to share information globally or among groups
  - Examples of AI functionality available in Lua:
    - AI.Signal
    - AI.FindObjectOfType
    - AI.GetAttentionTargetType (Visual, Memory, Sound, None)
    - AI.GetAttentionTargetAIType (Actor, Grenade, Car, etc.)
    - AI.GetRefPointPosition
    - AI.DistanceToGenericShape
    - AI.SetBehaviorVariable (to change behavior)
    - AI.CanMelee
    - AI.RecComment (make comment for Visual AI Debugger
- Scripting with XML
  - Behavior/coordination trees
  - AI communications
  - Items (e.g., weapons)
- Entity system
  - Spatial queries – GetPhysicalEntitiesInBox()
  - AI agents and vehicles are entities in the Entity system
  - To spawn an entity, its Entity class is required – Can be defined either using the `.ent` file in `Game\Entities` OR through a C++ call to RegisterFactory() in game code
  - An entity pool can be used to limit the number of active AI agents per each specified Entity class.
- AI Debugger and AI Debug Draw
  - Use AI Debugger to track multiple features, including active behavior, signal received, attention target, comments, etc.
  - `ai_DebugDraw`
    - 1 – Basic info on AI agents (selected by `ai_DrawAgentStats`)
    - 74 – All of the navigation graph (can be slow)
    - 79 – Parts of the navigation graph around the player
  - `ai_statsTarget <AI name>` – Detailed info for the specified AI
  - `ai_DebugTargetTracksAgent <AI name>` – Perception information on the specified AI
  - `ai_Recorder_Auto` – Record AI activity in Editor game mode for AI Debugger
  - `ai_DebugTacticalPoints` – Debug TPS queries
  - `ai_DrawPath <AI name>` – Draw the path of the specified AI (optionally specify "all" for all AI agents)
  - `ai_DrawPathFollower` – Draw the actual path being followed
  - `ai_DrawSmartObjects` – Display smart objects and their classes and attributes
  - `ai_DebugDrawEnabledActors` – List currently enabled AI agents.

Execution Context

- AI update is called every frame, but are fully updated only at ~10Hz
- Some AI subsystems use independent time-slicing (pathfinding, tactical point, dynamic waypoints updating, smart object, interest, and dead bodies removal)
- Some services can be called synchronously from game code (such as tactical point system (TPS) queries)

Pathfinding Costs

For agents to behave in a believable way, they need to find paths that are appropriate for their current state. Sometimes these paths will take the most direct route; other times they will be longer paths to maximize use of roads, cover, or other properties of the environment. The current system needs to be extended to support this. The pathfinding system uses A* to find minimal-cost paths.

The cost of a path is given by the sum of the costs of the links that make up that path. Currently the cost traversing a link in the navigation graph is normally simply the physical (3D) length of that link. However, the A* implementation makes it easy for the requester to modify these distance-based costs with simple code changes to extend the current system . For example, the cost of traveling between two road nodes can be scaled by a factor of 0.1 so that road-traveling agents have a strong preference for finding road-based paths.

The cost of a path link connecting two graph nodes should be determined by two sets of properties:

- Link properties, including the path's length.
- Pathfinding agent properties in relation to link properties. For example, a stealthy agent might evaluate a link passing through trees as a lower cost-per-unit-length than one passing along a road. However, the same agent might reach a different conclusion when leading a convoy containing vehicles.

In general, the cost of a link is determined by the product of these two factors: the link-length multiplied by a relative cost-per-unit-length. The latter is what needs to be determined.

Problem: Calculating Costs at Run Time

We want to use the same navigation graph for different kinds of agents. This means that link cost should be calculated at run time by combining the inherent link properties with the agent properties.

Link properties

Associate these properties with each link:

Link.Length
Length of the link (in meters).

Link.Resistance
The link's resistance to traversal. A road would be close to 0, off-road would be larger, water deep enough to require swimming might be close to 1.

Link.Exposure
How exposed the link is. Trees and dense vegetation would be close to 0, low vegetation would be larger, and a road/open space would be close to 1.

Link.DeepWaterFraction
Fraction of the link that contains deep water (e.g., > 1.5m).

Link.DestinationDanger
Additional "danger value" associated with the destination node. A dead body might be 1. This value can be stored in the destination node itself to save memory.

Agent properties

Associate these properties with each agent (normally set when the agent is created):

Agent.CanTraverseTriangular
True/false indicator determining if the agent can traverse triangular nodes.

Agent.CanTraverseWaypoint
True/false indicator determining if the agent can traverse waypoint nodes.

Associate these properties with an agent if relevant for the link type:

Agent.CanSwim
True/false indicator determining if the agent can swim.

Pathfinder request properties

Associate these properties with each agent pathfinder request:

Agent.TriangularResistanceFactor
Extra link cost factor when the link is of type Triangular and its resistance is 1.

Agent.WaypointResistanceFactor
Extra link cost factor when the link is of type Waypoint and its resistance is 1.

Agent.RoadResistanceFactor
Extra link cost factor when the link is of type Road and its resistance is 1.

Associate these properties with an agent pathfinder request if relevant for the link type (note: if a path link has different start/end node types, the result is obtained by averaging):

Agent.SwimResistanceFactor
Extra link cost factor when the link deep water fraction is 1.

Agent.ExposureFactor
Extra link cost factor when the link's exposure is 1.

Agent.DangerCost
Extra link cost when the link danger value is 1.

All link properties, except for Link.DestinationDanger, are calculated when the triangulation is generated. Link.DestinationDanger is initially set to 0 and then calculated as the game runs. For example, whenever a character dies, each link going into the death node will have its DestinationdangerCost incremented by 1. This will cause an agent with Agent.DangerCost = 100 to prefer paths up to 100m longer (assuming no other path cost differences) in order to avoid this death node. These link modifications need to be serialized to support load/save.

In addition, extra costs can be calculated at run time. For example, an extra cost associated with exposure could be added when an agent wishes to find a path that avoids the player; this can be done by using raycasts in the A* callback that calculates costs.

When determining pathfinding costs, there are two problems that need to be solved:

- How should the link properties be calculated?
- How should the link and agent properties be combined to give a numerical cost for traversing each graph link?

Keep in mind that link properties represent the average nature of the environment over the length of the link. If the region has not been triangulated reasonably finely, this may negatively impact the quality of pathfinding results. If the impact is significant, it may be necessary to add additional triangulation points.

An additional issue to consider: should pathfinding differentiate between variable visibility conditions, such as night vs. day or fog vs. clear weather? This would involve splitting the link exposure into terms derived from physical cover and shadow cover. Given the number of links involved, adding too much information of this type to each link should be considered carefully. A simpler solution might be to have stealthy agents be less likely to request a stealthy path in these conditions, or to set the agent's ExposureFactor lower.

Solution

Calculating link properties

Because link resistance is only dependant on the actual type of nodes involved in the link, it can be stored in a lookup table. Here's an example set of resistance values for each node type:

Node type	Resistance
Triangular-no-water	0.5
Triangular-water	1.0
Waypoint	0.6
Road	0
Flight/Volume	0

Note

- Consider adding a separate resistance for Flight/Volume in underwater terrain.
- For links between nodes of different types, the resistance values can be averaged.

The Link.Exposure value, which is stored in the link, is determined by the environment properties sampled over the length of the link. For triangular, waypoint and volume navigation regions, this can be done by firing rays from points along the link. (This is done by using IPhysicalWorld::RayWorldIntersection and checking for HIT_COVER | HIT_SOFT_COVER with COVER_OBJECT_TYPES.) It does not make sense to try to get a really accurate value, because in practice the beautified path will not follow the link directly.

Combining link and agent properties

Link cost must account for intersections between link properties and agent properties. For example: if a link is marked as containing deep water and the agent cannot swim, the link should be treated as impassable.

A factor representing the extra costs associated with travel resistance and exposure will be calculated, and the total link cost should be set as follows:

```
Cost = Link.DestinationDanger * Agent.DangerCost + (1 + Factor) * Link.Length
```

where

```
Factor = Agent.[link-type]ResistanceFactor * Link. [link-type]Resistance +
Agent.ExposureFactor * Link.Exposure
```

Consider this scenario: with no exposure/destination costs, and assuming that road links have Link.Resistance {{0}} while off-road links have Link.Resistance {{0.5}}, then in order to make road travel ten times more attractive than off-road (such as if the agent is a car), the agent could have Agent.TriangularResistanceFactor set to {{(10-1)/0.5}} (or 18) and Agent.RoadResistanceFactor set to 0.

If the agent is a human character that always moves at about the same speed whether or not it is on or off a road, then it could have both Agent.TriangularResistanceFactor and Agent.RoadResistanceFactor set to 0.

Assuming the agent can traverse deep water or is not affected by water (such as a hovercraft), Agent.SwimResistanceFactor could be set to 0. For a human agent, this factor might be set to a value as high as 3.0, so that the agent will take significant detours to avoid swimming across a river.

Sensory Models

Overview

This topic describes the modelling and principal operation of the sensors implemented in the Lumberyard AI system. These include the visual sensors, sound sensors, and a general-purpose signalling mechanism.

Sensory information is processed during a full update of each enemy (the actual time that a sensory event was received is asynchronous). These sensors are the only interface the enemy has with the outside world, and provide the data that the enemy will use to assess their situation and select potential targets. All sensors are completely configurable, and they can be turned on/off at run-time for any individual enemy.

Vision

The visual sensory model is the heart of the AI system. It is an enemy's most important sense. The model is designed to simulate vision as realistically as possible, while still maintaining a low execution cost, using a combination of compromises and optimizations.

During a full update for an individual enemy, the system traverses all potential targets from the enemy’s point of view and runs each one through a visibility determination routine. All targets that survive this filtering procedure are placed in a visibility list that is maintained until the next full update. For a target to persist as "visible" it must pass the visibility test in each full update. Targets that change from visible to not visible during an update are moved to a memory targets list. If a previously visible target becomes visible again, it is moved from the memory target list back to the visibility list. Memory targets have an expiration time to simulate the enemy "forgetting" the target; this time interval is determined by several factors, including the threat index of the target and the length of time it was visible. Visible targets are given the highest priority and will become the current attention target even if there is another target with a higher threat index. This approach simulates the natural tendency of humans to act based on what they see faster than on what they remember (or hear).

Visibility Determination

The visibility determination routine determines whether a target is considered visible to an enemy. It is run against each of the enemy's potential targets during a full update.

Identifying Targets

Visibility determination can be very CPU intensive; to mitigate this cost, only potential targets are evaluated for visibility. There is a mechanism to register any AI object as an object that should be included in the visibility determination (including user-defined objects). This includes objects such as the grenades in Lumberyard, flashlights, etc. There are also special objects called attributes, which will be discussed in more detail later in this topic.

To be considered a potential target, an AI object must be:

- currently active
- of a different species than the enemy (enemies don't need to keep track of members of their own team)

In addition, the visibility determination test is performed automatically against the player, even if the player is of the same species as the enemy. This rule ensures that the player is accurately specified as an object type and is always taken into account when checking visibility.

The game developer can also designate certain AI object types for visibility determination. These user-defined types are added to a list maintained by the AI system identifying object types to be included in the visibility check. Objects can be freely added to and removed from this list, even from script. To include an object in the list, specify an assessment multiplier to the desired object type. For example, refer to the file `aiconfig.lua`, which can be found in the `/scripts` folder. For more about assessment multipliers, see the topics on threat assessment.

Checking Visibility

Each potential target identified is evaluated for visibility using a series of tests. In situations where the player is facing a single species, no visibility determination is performed between AI enemy objects, only against the player.Key measures determining visibility include:

Sight-range test

This check is done first, as it is fast and cheap to filter out all AI objects that are outside the enemy's sight range. This is done by comparing the distance between enemy and target against the enemy's sight range value.

enemy sight range
: Floating point value that determines how far the enemy can see (in meters); the value represents the radius of a sphere with the enemy at the center.

Field-of-view test

Objects that are inside the enemy's sight range sphere are then checked for whether they are also inside the enemy's field of view (FOV).

enemy field of view
: Floating point value that determines the angle of the enemy's visibility cone (in degrees); the cone's tip is at the enemy's head and extends outward in the direction the enemy is facing.

The FOV angle determines how far to the left and right the enemy's peripheral vision allows them to see, and is relative to the direction the enemy is currently facing. The FOV angle value indicates the full range of vision; for example, with an FOV of 180 degrees, an enemy can see everything within 80 degrees from their forward orientation. The FOV check is performed using a simple dot product between the enemy's orientation vector and the vector created as the difference between the positions of the potential target and the enemy. The resulting scalar is then compared to the value of the FOV. Note that by using a conical shape, FOV is not limited to 2D representations.

Physical ray test

Objects that survive the two initial checks are very likely to be seen. The next check is an actual ray trace through the game world, which is an expensive process. Because the low layer of the AI system performs distributed updates over all frames, it is very seldom that a large number of rays needs to be shot per frame. Exceptions include scenes with a high number of objects belonging to different species and huge combat scenes, such as those with more than 20 participants per species.

The visibility physical ray is used to determine whether there are any physical obstacles between the enemy and the target. It originates from the head bone of the enemy character (or if the enemy does not have an animated character, it originates from the entity position – which is often on the ground) and is traced to the head bone of the target (if it has one, otherwise the entity position is used). If this visibility ray hits anything in its path, then the target is considered not visible. If the ray reaches the target without hitting any obstacles, then the target has passed this tests and can be added to the visibility list for this update.

Not all obstacles are the same. The physical ray test distinguishes between hard cover and soft cover obstacles. For more information on how cover type affects enemy behavior, see the section on soft cover later in this topic.

Perception test

This test is for player AI objects only (and other AI objects as defined by the game developer). Once the player has passed all the visibility tests for an enemy, this final test determines whether or not the enemy can see the player object. Each enemy calculates a perception coefficient for the player target, which ultimately describes the likelihood that the enemy can see the target.

perception coefficient (SOM value)
Floating point value (between 0 and 10) that defines how close the enemy is to actually seeing the target.

The perception coefficient is calculated based on a range of factors, including the distance between enemy and target, height of the target, and whether the target is moving. The value must reach the maximum value (currently 10) before it can receive a definite visual stimulus--that is, see the target.

For more details on how a perception value is derived, see the section on calculating perception later in this topic.

Soft Cover Visibility and Behavior

The physical ray test also evaluates the surface type of obstacles when determining visibility. The AI system can discriminate between two types of surfaces: soft cover and hard cover. The primary difference in a physical sense is that game players can pass through soft cover but cannot pass through had cover. Players can hide behind soft cover objects but the visibility determination is slightly “skewed” when a target is behind a soft cover object rather than a hard cover object or just in the open. When determining a target's visibility behind soft cover, the AI system takes into account whether or not the enemy already identified the target as "living" (not a memory, sound or other type of target). If the enemy does not have a living target, then the soft cover is considered equal to hard cover and normal visibility determination is performed. This occurs when the enemy is idle--or when the enemy is looking for the source of a sound but has not yet spotted it.

However, the behavior is slightly different when the enemy already has a target identified. During the physical ray test, if only soft cover is detected between the enemy and their target, then the target will remain visible for short length of time--between 3 and 5 seconds. If the target remains behind soft cover during this time, the enemy will eventually lose the target and place a memory target at the last known position. However, if the target leaves soft cover within this time, then the timer is reset and normal visibility rules are put into effect.

This behavior simulates the following example: when a soldier perceives that the target has run inside a bush, they do not immediately forget about it because they can make out the target’s silhouette even inside the bush. But following a target like that is difficult over time, and after a while the soldier will lose track of the target. The same rules apply to covers made of penetrable cover, like wood, but the rationale is a bit different. If a target runs behind a thin wooden wall, the soldier knows that bullets will still pierce the wall, so for a short time the target's position is still known, and the enemy continues to shoot through it. This can make for some really intense situations in a Lumberyard game.

In order for this process to work in a closed and rational system, all surfaces in the game need to be properly physicalized (wood, grass, and glass should be soft cover, while rock, concrete, metal should be hard cover). This is consistently done in Lumberyard.

Perception Calculation

Unlike visibility between AI agents, visibility of player objects to enemy AI agents in Lumberyard is not an on/off switch. This added layer of complexity is designed to allow for variations in game playing style (such as action versus stealth). Perception allows the player to make a certain number of mistakes and still be able to recover from them. (This is one of the reasons why a player AI object is specifically defined even in the lowest layer of the AI system hierarchy.) It is not used with other AI objects, where “switch” vision is used (that is, the target is visible as soon as a ray can be shot to its position). Note that it is possible to declare some AI objects should also trigger use of a perception coefficient.

An enemy is given a perception coefficient that describes how close the enemy is to actually seeing a particular target. The initial value of the perception coefficient is 0 and increases or decreases based on a defined set of rules. If a player target passes all prior visibility tests, the enemy begins applies the perception coefficient. Once the maximum value has been reached, the player target is visible to the enemy. This statement contains several corollaries:

- Each enemy has a perception coefficient for each player target it is processing.
- Each enemy will receive notification that the player target is visible only after the perception coefficient reaches maximum value.
- The perception coefficient of two different enemies are unrelated, even for the same player target.
- There is no game-level perception coefficient (that is, a value that determines how any enemy perceives a player target), although this information could be derived by statistics.

When an enemy starts receiving notification that a player target is passing the visibility determination routine, it begins calculating the perception coefficient. This is done by evaluating the following factors, all of which impact the rate at which the coefficient increases. Keep in mind that a player target must still pass all other phases of the visibility determination routine before the perception coefficient is applied.

Distance
Distance between the enemy and the player target has the highest influence on perception. The closer the player target is to the enemy, the faster the coefficient rises, while greater distances cause the coefficient to rise slower. The increase function is a basic quadratic function. At distances very close to the enemy, the time to reach maximum perception is almost non-existent and the target is instantly seen. In contrast, a player target may be able to move more freely along the boundaries of the enemy's sight range, as the perception value rises more slowly.

Height from ground
This factor takes into account the player target’s distance above the ground. The rationale for this behavior is that a prone target is much harder to spot than one who is standing upright. The AI system measures the distance of the target from the ground based on the “eye height” property of an AI object. This property is set when the AI object is initialized, and can be changed at any time during execution of the game. If enemies and players are represented in the game by animated characters, the eye height is calculated using the actual height of the character's head bone. This factor influences the rate of increase in the perception coefficient as follows: if the player target has a height above ground of less than one meter, the increase due to distance is lowered by 50%.

Target motion
The perception coefficient is affected by whether or not the player target is moving. Movement attracts attention, while stationary targets are harder to spot. This factor influences the rate of increase in the perception coefficient as follows: if the player target is standing still, the increase due to other factors is lowered by additional 50%.

Artificial modifiers
Additional values can define how fast the perception coefficient increases. Some affect all enemies in the game world, while some affect only particular targets. An example of a modifier that affects all enemies is the console variable `ai_SOM_SPEED`. Its default value varies depending on a game's difficulty level, but it provides a constant multiplier that is applied on top of all other calculations, and it applies to all enemies. In contrast, it is possible to set a custom multiplier for a specified object type

that is used only for certain player targets; however, this setting is limited to the lowest level of the AI system and is not available for tweaking.

The effect of perception is cumulative while the target is considered visible to the enemy. A floating point value is calculated based on the factors described above, and each time the enemy fully updated, this value is added to the perception coefficient (along with an updated visibility determination). So, for example, a player target that is within the enemy's range of sight might remain unperceived by the enemy significantly longer if they are crouching and motionless.

At the same time, a non-zero perception coefficient can fall back to zero over time if value is not increased constantly with each full update. For example, a player target might become visible for a few seconds, raise the coefficient up to 5, and then break visual contact. In this scenario, the coefficient will drop slowly to zero. This scenario was implemented to reward players that tactically advance and then pause before continuing; players can wait for the coefficient to drop to zero before continuing to sneak.

A statistical overview of the perception coefficients of all enemies for a player is used for the HUD stealth-o-meter, showing as a small gauge to the left and right of the radar circle in the HUD. It represents the highest perception coefficient of the player across all enemies that currently perceive him. In effect, it shows the perception coefficient of the one enemy that is most likely to see the player. so, a full stealth-o-meter does not mean that all enemies see the player; it means that there is at least one enemy that can. An empty stealth-o-meter means that currently no enemy can see the player.

Attribute Objects

An attribute object is not a full AI object; instead, it is more of a special helper that can be attributed to an existing AI object. The attribute is a special class of AI object, specifically defined at the lowest level in the AI system. Every attribute object must have a principal object associated with it. The principal object can be any type of an object (including puppet, vehicle, player, etc,.) but cannot be an attribute.

Attributes can impact visibility determination. When an enemy determines that it sees an attribute object, the system will switch the attribute with the principal object before adding it into the visibility list of the enemy. Thus, an enemy who sees an attribute will believe it is seeing the principal object attached to the attribute.

Essentially, attributes are a systematic way of connecting certain events to a single target. For example, a player switches on a flashlight and the beam hits a nearby wall. The light hitting the wall creates an attribute object associated with the principal object, which is the player. In this scenario, the player is hidden, but because an enemy sees the attribute object (the light on the wall), it will in fact "see" the player. The rationale is that enemies have enough intelligence to interpolate the origin of the light ray and thus know the player’s position.

This feature is also used with regard to rocks, grenades, rockets etc. It can be extended to add more features to a game; for example, a target might leave footprints on the ground that evaporate over time. The footprints spawn attribute objects, which enable any enemy who sees them to perceive the location of the target who left them. Another application might be blood tracks.

To ensure that attribute objects are included in the visibility determination, they must have an assessment multiplier set. Refer to `aiconfig.lua` in the `Scripts\AI` folder to see where the AI system defines the multiplier for attribute objects.

Flight

Use these archetypes and flow nodes in conjunction with entities to control flying vehicles. See these archetypes in the characters archetype library:

- **CELL/Helicopter.Regular**
- **Ceph/Dropship.Regular**

- **Ceph/Gunship.Regular**

The following flow nodes to be used with these entities are found under the Helicopter category.

FollowPath

This flow node sets the current path that the flight AI uses.

- When the AI is not in combat mode.
  - If the AI is set to loop through the flow node path, the AI tries to go from its current location to the closest point of the path and then follows it to the end. The node outputs indicating that the AI has reached the end of the path is sent once only.
  - Without looping, the AI tries to go from its current location to the beginning of the path and then follows it to the end.
- When the AI is in combat mode.
  - If the target is visible, the path is used to position the AI in the best location to attack the target. It is also used to navigate between positions within the path.
  - If the target is not visible, the path is used as a patrol path. Where possible, it simplifies setup to have paths in combat mode be set to loop.

EnableCombatMode

This flow node enables or disables the AI's ability to position itself within the combat path in order to engage and shoot at its current target. By default, an AI is not in combat mode until it's explicitly allowed to go into combat mode.

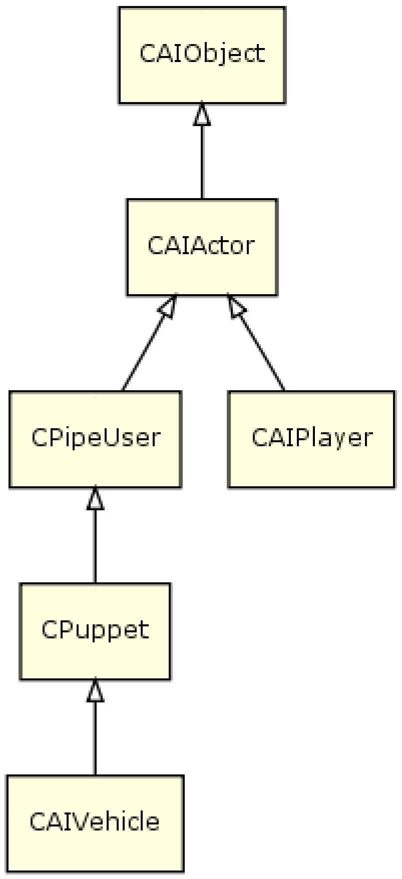
- When an AI is in combat mode and has identified a target location, it will try to reposition itself within the current path to a position from which it can engage.
- Any character of an opposing faction is a good candidate for a target.

EnableFiring

This flow node enables or disables the ability of the AI to shoot at its current target when in combat mode. By default, an AI is allowed to fire when in combat mode until it's explicitly disabled using this node.

AI C++ Class Hierarchy

C++ classes for AI objects are structured as follows.



CAIObject
: Defines basic AI object properties (entity ID, position, direction, group ID, faction, etc.)

CAIActor
: Basic perception and navigation, behavior selection, coordination, blackboard, AI territory awareness, AI proxy

CAIPlayer
: AI system's representation of an actual game player

CPuppet
: Aiming, firing, stances, covers, a full-fledged AI agent

CAIVehicle
: Vehicle-specific code

AI System Concept Examples

This section includes the following topics:

AI Multi-Layered Navigation

Useful AI debug draw:

- ai_useMNM=1
- ai_DebugDraw=1
- ai_DebugDrawNavigation=1
- ai_DrawPath=all
- ai_DrawPathFollower=1

Individual AI: Dynamic Covers

Example: CoverSurface and HMMWV

This example shows the use of dynamic covers that are generated offline and adjusted during run time.

Useful AI debug draw:

- ai_DebugDraw=1
- ai_DebugDrawCover=2
- [AI/Physics] is on

Individual AI: Tactical Points

Example: A very shy civilian who always wants to hide from the player

- Tactical point system (TPS) query:

```
AI.RegisterTacticalPointQuery({
    Name = "Civilian_HideFromEnemy",
    {
        Generation =
        {
            cover_from_attentionTarget_around_puppet = 25
        },
            Conditions =
        {
            reachable = true,
        },
        Weights =
        {
            distance_from_puppet = -1,
        },
    },
});
```

- Useful AI debug draw:
  - ai_DebugTacticalPoints=1
  - ai_StatsTarget=Grunt1
  - ai_TacticalPointsDebugTime=10
- For more realism, add the following before goalop TacticalPos:

```
<Speed id="Sprint"/>
```

Group and Global AI: Factions

Example: AI formations of different factions

Place on a map three grunts of the following factions. Note who is hostile to who.

- grunts
- assassins
- civilians

For example:

```
<Factions>
    <Faction name = "Players">
        <Reaction faction- "Grunts" reaction="hostile"/>
        <Reaction faction- "Civilians" reaction="friendly"/>
        <Reaction faction- "Assassins" reaction="hostile"/>
    </Faction>
    <Faction name="Civilians default="neutral"/>
    ...
</Factions>
```

(see `Game/Scripts/AI/Factions.xml`)

Group and Global AI: Flow Graphs

Flow Graph Editor allows non-programmers to build global AI logic visually. Experiment with flow graph debugger features, such as signal propagation highlighting and breakpoints.

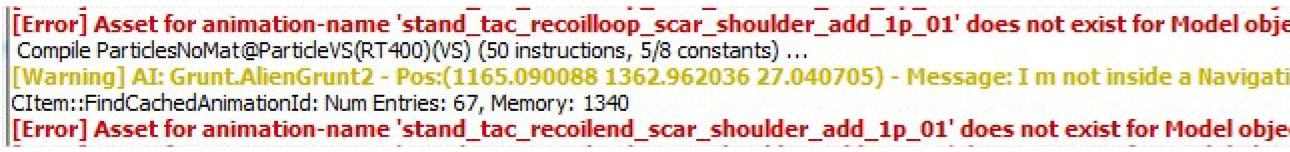
AI Bubbles System

The AI Bubbles system collects AI error messages for level designers to address. This system streamlines the debugging process by helping to track down which system(s) are connected to a problem. To use the AI Bubbles system, programmers need to push important messages into the system, which will then provide notification to the level designers when a problem is occurring.

Message Display Types

Message include a set of information (agent name, position, etc.) that help the designer to understand that something is wrong in the normal flow. Message notifications can be displayed in any of the following ways:

- Speech bubble over the AI agent
- Error message in the console



```
[Error] Asset for animation-name 'stand_tac_recoilloop_scar_shoulder_add_1p_01' does not exist for Model obje
 Compile ParticlesNoMat@ParticleVS(RT400)(VS) (50 instructions, 5/8 constants) ...
[Warning] AI: Grunt.AlienGrunt2 - Pos:(1165.090088 1362.962036 27.040705) - Message: I m not inside a Navigati
CItem::FindCachedAnimationId: Num Entries: 67, Memory: 1340
[Error] Asset for animation-name 'stand_tac_recoilend_scar_shoulder_add_1p_01' does not exist for Model obje
```

- Blocking Windows message box

Specifying Notification Display Types

Use one of the following ways to specify a display type for error messages:

Console

ai_BubblesSystem
Enables/disables the AI Bubbles System.

ai_BubblesSystemDecayTime
Specifies the number of seconds a speech bubble will remain on screen before the next message can be drawn.

ai_BubblesSystemAlertnessFilter
Specifies which notification types to show to the designer:

- 0 - No notification types
- 1 - Only logs in the console
- 2 - Only bubbles
- 3 - Logs and bubbles
- 4 - Only blocking popups
- 5 - Blocking popups and logs
- 6 - Blocking popups and bubbles
- 7 - All notifications types

ai_BubblesSystemUseDepthTest
Specifies whether or not the notification needs to be occluded by the world geometries.

ai_BubblesSystemFontSize
Specifies the font size for notifications displayed in the 3D world.

C++

In C++, use the method AIQueueBubbleMessage() to define how to display the message notification.

Method signature:

```
bool AIQueueBubbleMessage(const char* messageName, const IAIObject* pAIObject,
 const char* message, uint32 flags);
```

Parameters:

messageName
String describing the message. This is needed to queue the same message error more than once. (The message can be pushed into the system again when it expires is deleted from the queue.)

pAIObject
Pointer to the AI object that is connected to the message.

message

Text of the message to be displayed.

flags

Notification type. This parameter can include one or more flags; multiple flags are separated using a pipe (|).

- eBNS_Log
- eBNS_Balloon
- eBNS_BlockingPopup

Example:

```
AIQueueBubbleMessage("COPStick::Execute PATHFINDER_NOPATH non continuous",
pPipeUser, "I cannot find a path.", eBNS_Log|eBNS_Balloon);
```

Lua Script

```
local entityID = System.GetEntityIdByName("Grunt.AlienGrunt1");
                AI.QueueBubbleMessage(entityID,"I cannot find a path.");
```

AI Tactical Point System

The Tactical Point System (TPS) provides the AI system with a powerful method of querying an AI agent's environment for places of interest. It includes the GetHidespot functionality and expands on the "hide" goalop.

TPS is a structured query language over sets of points in the AI's world. Using TPS, AI agents can ask intelligent questions about their environment and find relevant types of points, including hidespots, attack points, and navigation waypoints. The TPS language is simple, powerful, and designed to be very readable.

For example, this query requests all points that match the following criteria:

- Generate locations within 7 meters of my current location where I can hide from my attention target.
- Only accept places with excellent cover that I can get to before my attention target can.
- Prefer locations that are closest to me.

```
hidespots_from_attentionTarget_around_puppet = 7
coverSuperior = true, canReachBefore_the_attentionTarget = true
distance_from_puppet = -1
```

TPS uses a highly efficient method to rank points, keeping expensive operations like raycasts and pathfinding to an absolute minimum. Queries are optimized automatically.

This section includes the following topics:

Tactical Point System Overview

Key features of the Tactical Point system (TPS) include:

- Use of a structured query language
  - Powerful and quick to change in C++ and Lua
- Query support for a variety of point characteristics, beyond conventional hiding places behind objects:
  - Points near entity positions
  - Points along terrain features
  - Points suggested by designers
  - Arbitrary resolutions of nearby points in the open or on terrain
- Query combinations, such as:
  - "Find a point somewhere behind me AND to my left, AND not soft cover, AND not including my current spot"
  - "Find a point hidden from my attention target AND visible to the companion"
- Preferential weighting, such as:
  - Find a point nearest to (or furthest from) a specified entity
  - Balance between points near an entity and far from the player
  - Prefer points in solid cover over soft cover
- Query fallback options, such as:
  - Prioritize good cover nearby; if none exists, go backwards to any soft cover
- Query visualization:
  - See which points are acceptable and which are rejected, as well as their relative scores
  - See how many expensive tests are being used by a query and on which points
- Automatic query optimization
  - Understands the relative expense of individual evaluations comprising queries
  - Dynamically sorts points based on potential fitness, according to weighting criteria
  - Evaluates the "fittest" points first, in order to minimize the use of expensive tests
  - Recognizes when the relative fitness of a point indicates that it can't be beat, in order to further reduce evaluations
  - Provides framework for further optimization specific to architecture, level, or locale

In addition to these key feature benefits, this framework offers these advantages from a coding perspective:

- Separates query from action
  - Arbitrary queries can be made at any time without changing the agent's state
- Query language is easy to expand
- Easily adapted for time-slicing (and in principle multi-threading):
  - Progression through query is fine-grained
  - Progress is tracked as state, so it can be easily paused and resumed
- Provides mechanism for delaying expensive validity tests on generated points until needed

TPS Query Execution Flow

The following steps summarize the definition and execution stages of a TPS query. Note that only stages 3 and 4 have a significant impact on performance.

1. Parse query:
   - Parse query strings and values.
   - This step is usually performed once and cached.
2. Make query request:
   - Query is made using C++, ScriptBind, goalops, etc.
   - A query is stateless; it does not imply a movement operation.
3. Generate points:
   - Create a set of candidate points.
   - Point candidates are based on the query's Generation criteria.
4. Evaluate points (this is by far the most intensive stage):
   - Accept or reject points based on Conditions criteria.
   - Assign relative scores to points based on Weights criteria.
5. Consider query fallbacks:
   - If no point matches the Conditions criteria, consider fallback options.
   - Where there is a fallback, return to step 3.
6. Visualize points:
   - If visualization is required, draw all points to screen.
   - Include point data such as its accepted/rejected status and relative scores.
7. Return results:
   - Return one or more points, if any fit the query conditions.
   - Each point is returned as a structure that describes the selected point.

There are some optimizations possible that depend on the execution flow. For example, relevant query results can be cached between fallback queries.

TPS Querying with C++

These C++ interfaces allow you to use TPS from other C++ code and within goalops. Lua queries are translated through it.

There are two C++ interfaces:

- Internal - For use only within the AI system
  - Uses a CTacticalPointQuery object to build queries
  - Allows you to create or adapt queries on the fly
  - Provides greater access to relevant AI system classes
- External - For use from any module
  - Suitable for crossing DLL boundaries
  - Simpler, not object-oriented, just as powerful
  - Uses stored queries for efficiency

Internal Interface Syntax

In the example below, some parsing is obviously taking place here. This is crucial to the generality of the system.

```
// Check for shooter near cover using TPS
static const char *sQueryName = "CHitMiss::GetAccuracy";
ITacticalPointSystem *pTPS = gEnv->pAISystem->GetTacticalPointSystem();
int iQueryId = pTPS->GetQueryID( sQueryName );
if ( iQueryId == 0 )
{
    // Need to create query
    iQueryId = pTPS->CreateQueryID( sQueryName );
   pTPS->AddToGeneration( iQueryId, "hidespots_from_attentionTarget_around_pup
pet", 3.0f);
    pTPS->AddToWeights( iQueryId, "distance_from_puppet", -1.0f);
}
pTPS->Query( iQueryId, CastToIPuppetSafe( pShooter->GetAI() ),vHidePos, bIsVal
idHidePos );
```

TPS Syntax Examples

The following examples and explanations illustrate the use of TPS query syntax. For a more detailed discussion of the TPS query language, see the topic on TPS Query Language Syntax and Semantics.

`option.AddToGeneration("hidespots_from_attTarget_around_puppet", 50.0)`

This query request is expressed as generation criteria and specifies a float to represent distance. The query is broken up into five words:

- "hidespots" indicates that generated points should positioned behind known cover as is conventional
- "from" and "around" are glue words to aid readability
- "target" specifies the name of the object to hide from
- "puppet" identifies a center location that points will be generated around
- The float value indicates the radial distance, measured from the center location, that defines the area within which points should be generated

Note that no raycasts are performed at this stage. We have here considerable flexibility, for example, how we choose to hide from a player: (1) somewhere near the player, (2) somewhere near us, or (3) somewhere near a friend. We can also specify a completely different target to hide from, such as an imagined player position. By providing flexibility at the point generation stage, we can support more powerful queries and allow users to focus computations in the right areas.

`option2.AddToConditions("visible_from_player",true)`

This query request is expressed as condition criteria, so we can expect a Boolean result. The query specifies points that are visible to the player, which is curious but perfectly valid. The term "visible" specifies a ray test, with "player" specifying what object to raycast to from a generated point.

`option2.AddToConditions("max_coverDensity",3.0)`

This query is expressed as a condition, so we can expect a Boolean result. The term "Max" specifies that the resulting value must be compared to the given float value--and be lower than. The term "coverDensity" identifies this as a density query (measuring the density of things like cover, friendly AI agents, etc.) and specifies measurement of covers.

`option1.AddToWeights("distance_from_puppet",-1.0)`

This query is expressed as a weight component; the query result will be a value between zero and one (normalized as required). Boolean queries are allowed to indicate preference (such as primary cover over secondary cover), with return values of 0.0 for false and 1.0 for true.

This query component indicates a preference for points at a certain location relative to an object. The term "distance" identifies this as a distance query, with the given float values specifying the distance amount. The term "puppet" identifies the object to measure the distance from.

TPS Querying with Lua

In Lua, there are two ways to use the TPS:

- Scriptbinds allow you to use TPS queries from a Lua behavior and have the results returned as a table without any side-effects. This can be useful for higher-level environmental reasoning, such as:
  - Choose behaviors based on suitability of the environment (for example, only choose a "sneaker" behavior if there's lots of soft cover available).
  - Run final, very specific tests on a short list of points, rather than adding a very obscure query to the TPS system.
  - Enable greater environmental awareness (for example, tell me three good hidespots nearby, so I can glance at them all before I hide).
- With goal pipes, you can use goalops to pick a point and go there, using a predefined TPS table:
  - Use a "tacticalpos" goalop, which is equivalent to a previous "hide" goalop.
  - Use fallback queries to avoid lists of branches in goalpipes.
  - More flexible goalops can be provided to decouple queries from movement.

Both methods define query specifications using the same table structure, as shown in the following example:

```
Hide_FindSoftNearby =
{
  -- Find nearest soft cover hidespot at distance 5-15 meters,
  -- biasing strongly towards cover density
  {
    Generation= {   hidespots_from_attentionTarget_around_puppet = 15 },
    Conditions= {   coverSoft = true,
            visible_from_player = false,
            max_distance_from_puppet = 15,
            min_distance_from_puppet = 5},
    Weights =   {   distance_from_puppet = -1.0,
    coverDensity = 2.0},
  },
  -- Or extend the range to 30 meters and just accept nearest
  {
    Generation ={   hidespots_from_attentionTarget_around_puppet = 30 },
    Weights =   {   distance_from_puppet = -1.0}
  }
}
AI.RegisterTacticalPointQuery( Hide_FindSoftNearby );
```

Note
Registering a query returns a query ID that then refers to this stored query.

Querying with Scriptbind

The following script runs a query using an existing specification. See comments in `Scriptbind_AI.h` for details.

```
AI.GetTacticalPoints( entityId, tacPointSpec, pointsTable, nPoints )
```

Querying with Goalops

The following script runs an existing query. Because queries can have fallbacks built in, branching is usually unnecessary (the branch tests are still supported).

```
AI.PushGoal("tacticalpos",1, Hide_FindSoftNearby);
```

TPS Query Language Reference

There are ways to define a query in both C++ and Lua (and potentially in XML), but the same core syntax is used. This page formally defines the TPS query language, with query components expressed in Generation, Conditions or Weights, and defines and discusses the query language semantics.

Query Syntax

Note
Non-terminal symbols are in bold. Not all of the symbols are implemented, but are shown for illustration.

```
Generator ::= GenerationQuery '_' 'around' '_' Object
Condition ::= BoolQuery | (Limit '_' RealQuery)
Weight    ::= BoolQuery | (Limit '_' RealQuery) | RealQuery
GenerationQuery ::= ( 'hidespots' '_' Glue '_' Object)
                      | 'grid' | 'indoor'
BoolQuery ::= BoolProperty | (Test '_' Glue '_' Object)
BoolProperty ::= 'coverSoft' | 'coverSuperior' | 'coverInferior' | 'currently
UsedObject' | 'crossesLineOfFire'
Test ::= 'visible' | 'towards' | 'canReachBefore' | 'reachable'
RealQuery = ::= RealProperty | (Measure '_' Glue '_' Object)
RealProperty ::= 'coverRadius' | 'cameraVisibility' | 'cameraCenter'
Measure ::= 'distance' | 'changeInDistance' | 'distanceInDirection' | 'dis
tanceLeft' | 'directness' | 'dot' | 'objectsDot' | 'hostilesDistance'
Glue ::= 'from' | 'to' | 'at' | 'the'
Limit ::= 'min' | 'max'
Object ::= 'puppet' | 'attentionTarget' | 'referencePoint' | 'player'
        | 'currentFormationRef' | 'leader' | 'lastOp'
```

Query Semantics

Note

- "Tunable" denotes that the exact values used should be possible to tweak/tune later.
- "Real" means that it returns a float value (rather than a boolean).

Objects

puppet
AI agent making a query

attentionTarget
Object that is the target of the AI agent's attention

referencePoint
AI agent's point of reference, perspective

player
Human player (chiefly useful for debugging and quick hacks)

Glue

from | to | at | the
Glue words used for readability in a query statement. Each query must have a glue word, but it has not active function and the parser doesn't distinguish between them. Readability is encouraged to aid in debugging and long-term maintenance.

Generation

Hidespot
Individual point just behind a potential cover object with respect to a "from" object (as in "hide from object"). There is typically one point per cover object. Use of this symbol should generate multiple points behind large cover objects and cope with irregularly shaped and dynamic objects.

Around
A glue word with special meaning. This word should be followed by the name of an object around which to center the generation radius.

Conditions/Weight Properties (use no object)

These properties relate to a specified point:

coverSoft
Boolean property, value is true if the specified point is a hidespot using soft cover.

coverSuperior
Boolean property, value is true if the specified point is a hidespot using superior cover.

coverInferior
Boolean property, value is true if the specified point is a hidespot using inferior cover.

currentlyUsedObject
Boolean property, value is true if the specified point is related to an object the puppet is already using (such as the puppet's current hide object).

coverRadius
Real (float) property, representing the approximate radius of the cover object associated with the specified point, if any, or 0.0 otherwise. When used for condition tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-5.0m] to [0.0-1.0]. (Tunable)

coverDensity
Real property, representing the number of potential hidespots that are close by to the specified point. When used for condition tests, returns an absolute value representing an estimate of the number of hidespots per square meter using a 5m radius sample. When used as a weight, returns a normalized value, mapping the range (0.0-1.0) to [0.0-1.0] (hidespots per square meter). (Tunable)

Conditions/Weight Test/Measures (require object)

These properties relate to a specified object, such as distance_to_attentionTarget or visible_from_referencePoint.

distance
Real (float) measure, representing the straight-line distance from a point to the specified object. When used for condition tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-50.0m] to [0.0-1.0]. (tunable)

changeInDistance
Real (float) measure representing how much closer the puppet will be to the specified object if it moves to a certain point. Takes the distance to the specified object from the current location and subtracts it from the distance to the object from the proposed new location. When used for condition tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-50.0m] to [0.0-1.0]. (tunable)

distanceInDirection
Real (float) measure representing the distance of the point in the direction of the specified object. Takes the dot product of the vector from the puppet to the point and the normalized vector from the puppet to the object. When used for tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-50.0m] to [0.0-1.0]. (tunable)

directness
Real (float) measure representing the degree to which a move to a certain point approaches the specified object. Takes the difference in distance to the object (changeInDistance) and divides it by the distance from the puppet to the point. Always uses the range [-1.0 to 1.0], where 1.0 is a perfectly direct course and negative values indicate movement further away from the object.

Limits

min | max
Limits can be used to test a real value in order to product a Boolean. Useful for conditions that can also be used as coarse Weights; for example, the condition MAX_DISTANCE = 10 can be used to express that a distance of less than 10 is preferable, but without favoring nearer points in a more general way.

Failing Queries

There are a few different ways queries can fail, and it's important to understand how each case is handled.

- **No points matched the conditions of the query.** This is a valid result, not a failure; the AI can move to fallback queries or try a different behavior.
- **The query does not make sense in the context of an individual point.** Sometimes a query doesn't make sense for a certain point or at a certain time. In this case, the query tries to return the "least surprising" results. For example: a query about a point generated in the open asks "is this soft cover?" The result will be "false", because this isn't any kind of cover. Query names should be chosen carefully to help avoid potential confusion.
- **The query does not make sense in the context of the puppet, at this time and for any point.** As with the point context issue, the query tries to return the "least surprising" results. For example: a query about a puppet asks "am I visible to my attention target?" when the puppet doesn't have an attention target. The query could return false, but it would disqualify every point. This case will usually indicate a code error--the puppet should have an attention target at this point, but does not. Note: This situation can cause a similar problem in the point generation phase, with a query like "generate hidespots from my attention target". both of these situations are flagged as code errors.

- **The query failed due to a code error.** You can test for errors in the TPS queries and raise them there also. For example, a query or combination hasn't been fully implemented yet or is being used as a kind of assert to test variables.

Point Generation and Evaluation

An AI agent makes a TPS point generation query in order to generate a set of points for consideration. Once generated, each point can be evaluated based on its position and any available metadata.

Generating Points

Input

The following information are required to generate points:

- Specific criteria defining the types of points to generate.
- A central or focal position around which to generate points. This might be the current position of the puppet itself, an attention target, or some other given position.
- For some queries, the position of a secondary object, such as a target to hide from.

It is possible to specify multiple sets of point generation criteria. For example, a query might request point generation around both the puppet and an attention target.

Processing

Based on the input, TPS begins generating points to evaluate. All points fall into two main types:

- Hidepoints. These are generated based on the following calculations:
  - Hideable objects
  - Generated only if a position to hide from was provided
  - Hidepoints represent final positions, for example calculating positions behind cover
  - Using the object and delaying finding an actual point is a possibility
- Open points. These are generated based on query specifications and the following calculations:
  - Usually on terrain, but may be on surfaces, etc.
  - Resolution/pattern (such as triangular with 1-meter spacing)
  - Potentially may perform more general sampling to find an exact point, but an initial resolution is still required
  - Radial/even distributions

Output

The result of a point generation query is a list of point objects. Each point object includes the point's position and available metadata, such as any associated hide objects.

Evaluating Points

Once a generation query generates a set of points, they can be evaluated. Point evaluation tries to establish the "fitness" of each point, that is, how well the point matches the specified criteria. The goal is to choose either one good point, or the best *N* number of good points.

Input

The following elements are required to evaluate points:

- List of candidate points from the point generator
- Point evaluation criteria:
  - Boolean – Condition criteria used to include or exclude a point independently of any other points
  - Weight – Criteria that, combined, give a measure of fitness relative to other points (those included by the boolean criteria)

Processing

The primary goal is to find an adequately good point as quickly as possible. Often, "adequately good" also means "the best", but there is a lot of potential for optimization if a user-specified degree of uncertainty is allowed.

The order of evaluation has a non-trivial and crucial impact on query efficiency. As a result, evaluation uses the following strategy to minimize the number of expensive operations:

1. Cheap booleans, with an expense on the order of one function call or some vector arithmetic. These allow the system to completely discount many points without significant cost. For example: *Is this point a primary or secondary hidespot? Is this point less than 5 meters from the target?*
2. Cheap weights, with an expense similar to cheap booleans. These allow the system to gauge the likelihood that a given point will eventually be the optimal choice; by focussing on points with a high likelihood, the number of expensive tests can be reduced. For example: *closeness_to_player * 3 + leftness * 2*.
3. Expensive booleans, approximately 100 times costlier. These are yes/no questions that will require expensive calculations to answer, but further eliminate points from contention. For example, the question *Is this point visible by the player?* requires an expensive ray test.
4. Expensive weights, with an expense similar to expensive booleans. These help to rank the remaining points. For example: *nearby_hidepoint_density * 2*

Algorithmic Details

It turns out that the system can go further with this by interleaving the final two steps and making evaluation order completely dynamic. Unlike conditions (booleans), weights don't explicitly discount points from further evaluation. However, by tracking the relative "fitness" of points during evaluation, we can still employ weights to dramatically reduce evaluations by employing two basic principles:

- Evaluate points in order of the their maximum possible fitness, to fully evaluate the optimal point as quickly as possible.
- If, based on the initial weight evaluation, a point can be established as better than any other point, then immediately finish evaluating it against the remaining conditions. If the point passes all condition criteria, then it is the optimal point and no other points need to be evaluated. In addition, this point does not need to be evaluated on any remaining weights.

This implementation is based on a heap structure that orders points according to their maximum possible fitness and tracks the evaluation progress of each point separately. Each weight evaluation collapses some of the uncertainty around the point, adjusting both the minimum and maximum possible fitness. If the weight evaluation scored highly, the maximum will decrease a little and the minimum increase a lot; if it scored badly, the maximum will decrease a lot and the minimum increase a little.

In each iteration, the next most expensive evaluation is done on the point at the top of the heap, after which the point is re-sort into place if necessary. If all evaluations on a point have been completed and it still has the maximum possible fitness, then it must be the optimal point. This approach tends towards evaluation of the optimal point with relatively few evaluations on all other points.

Output

The result of point generation evaluation is a single point or group of *N* number of points, and the opportunity to request all metadata leading to its selection. As a result, behaviors can adapt their style to reflect the nature the hidepoint received.

Integration with the Modular Behavior Tree System

From inside the Modular Behavior Tree (MBT), the **<QueryTPS>** node can be used to call pre-defined TPS queries by name. The **<QueryTPS>** node will return either success or failure.

The most common usage pattern involving the **<QueryTPS>** node is to use it in conjunction with the **<Move>** node inside a **<Sequence>** to determine the status of a specified position. The example below illustrates a call to a pre-defined TPS query called **SDKGrunt_TargetPositionOnNavMesh**, with the expected inputs. If the query succeeds, the AI agent will move to the queried position.

```
<Sequence>
    <QueryTPS name="SDKGrunt_TargetPositionOnNavMesh" register="RefPoint"/>
    <Move to="RefPoint" speed="Run" stance="Alerted" fireMode="Aim" avoid
Dangers="0"/>
</Sequence>
```

The definition of the pre-defined query **SDKGrunt_TargetPositionOnNavMesh** is as follows.

```
AI.RegisterTacticalPointQuery({
    Name = "SDKGrunt_TargetPositionOnNavMesh",
    {
        Generation =
        {
            pointsInNavigationMesh_around_attentionTarget = 20.0
        },
        Conditions =
        {
        },
        Weights =
        {
            distance_to_attentionTarget = -1.0
        },
    },
});
```

Future Plans and Possibilities

The following topics represent potential areas of development for TPS.

Higher-level environmental reasoning

One possible application of TPS: rather than simply using TPS to choose a point and move to it, there is the potential for some nice environmental deductions based on results.

For example: The player runs around a corner, followed by an AI puppet. When the AI puppet turns the corner, the player is no longer visible. The puppet queries TPS for places it would choose to hide from itself, with the following possible results.

- TPS returns that 1 hidepoint is much better than any other. This is because there's a single large box in the middle of an empty room. The AI puppet assumes the player is there and charges straight at the box, firing.
- TPS returns that there are several good hiding places. This is because there's a stand of good cover trees. All the hidepoints are stored in a group blackboard, and the AI puppet (or a group) can approach each spot in turn to discover the player.

This scenario is workable with some extra code, and much easier when built upon TPS.

Sampling methods

When generating points in the open, generate points in a grid or radially around objects and treat each point individually. This supports a basic sampling method. Where an area must be sampled, some kind of coherency in the evaluation functions can be assumed, and so could use some adaptive sampling approaches instead.

Dynamic cost evaluation

A crucial aspect of optimizing TPS involves adjusting the relative expense function of queries. The costs of evaluations will vary across platforms, levels, and even locations within levels, and will change over time as the code changes. It is critical to make sure that the evaluation order is correct, to prevent more expensive evaluations from being favored over cheaper ones. The need to profile the evaluation function in all these difference circumstances suggests an automatic profiling solution at run-time.

In addition, the relative weighting of weight criteria should also be considered; a cheap query may not be worth doing first if it only contributes 10% of the final fitness value, while an expensive query that contributes 90% may actually save many other evaluations.

Relaxing the optimality constraint

When evaluating points the maximum and minimum potential fitness is always known at every stage; this provides the error bounds, or a relative measure of uncertainty about the point.

It may make sense to relax the optimality constraint and accept a point when it becomes clear that no other point could be significantly better. For example, the minimum potential fitness of a point may be less than 5% lower than the maximum potential fitness of the next best point. This information could be used to stop evaluation early and yield a further performance saving.

Navigation Q & A

Big Triangles and Small Links Between Them

Q: I have created a big flat map, placed an AI agent on it, and generated AI navigation triangulation. I noticed that the AI agent doesn't always take the shortest straight path from point A to point B. Why?

A: To illuminate the issue, use the following tools:

- AI debug console variable `ai_DebugDraw` set to "74". This value draws the AI navigation graph. (Note: a value of 79 will run faster, but limits the result to the area close to the player (with 15 m).
- AI debug console variable `ai_DrawPath` set to "all". This variable draws AI agent paths, including links (the corridors between adjacent triangles).
- The **Ruler** tool in Editor, used to visualize paths. You don't even need actual AI agents on the map to run experiments. (Note: this tool is located between **Snap Angle** and **Select Object(s)**.)

The AI navigation triangulation is intended to be fast and have a small memory footprint. One of the decisions made in this regard was to use 16-bit signed integers to store corridor (or "link") radius measurements between two adjacent triangles. Using centimeters as the uint of measure, this means that the maximum link radius is 32767 cm (327.67 m). When an AI agent moves to another triangle, it can only go through this corridor, which is naturally very narrow if the triangles are still very large. This problem does not exist for triangles with edges less than 2 * 327.67 = 655.34 m.

This problem can only appear in the very initial stages of map development. Every forbidden area, tree or other map irregularity makes triangulation more developed, which results in more triangles that are smaller in size. As a result, the problem goes away.

Path Following

Q: How does path following actually work? Where to start?
A: See the topic on Path Following (p. 31).

Auto-Disabling

Q: How do you keep patrols always active, regardless of their distance from the player?
A: See the topic on Auto-Disable (p. 34).

Path Following

This topic provides some high-level insight on how path following is done in Lumberyard. To illustrate some concepts, we'll use the relatively simplistic example of Racing HMMWVs, which is a good representation of classic path following as presented in many AI texts.

Path following with Racing HMMWVs adheres to the following sequence.

1. Get the closest (to the AI agent) point on path.
2. Get the path parameter of this point. Paths usually have some kind of parametrization, *t -> (x,y,z).*
3. Add a certain value, usually called a "lookahead", to this parameter.
4. Get the path point that corresponds to this new parameter. This is called the look-ahead position.
5. Use this point as the navigation target.
6. If the vehicle is stuck, beam it straight to the closest point on the path.

Goalop "Followpath"

Use the goalop *followpath* to instruct an AI agent to follow a path. You can observe this sequence in action by setting a breakpoint at the beginning of a call to COPFollowPath::Execute. In the call stack window in Visual Studio, you'll be able to see the update operations for all (active) AI agents being called as part of the AI system update procedure. This action in turn calls the execute operations of the currently active goalops being run by the AI.

COPFollowPath::Execute accomplishes the following tasks:

- Uses the goalop *pathfind* to find a path leading to the beginning of a path. Optionally, it finds a path to the closest point on a path using a parameter passed to the *followpath* goalop.
- Traces the path by following it using the goalop *trace*
- Listens for the signal "OnPathFollowingStuck" to make sure the AI agent isn't stuck

The goalops *pathfind* and *trace* are commonly used for navigational goalops, including *approach* and *stick*.

COPTrace::ExecuteTrace and COPTrace::Execute

COPTrace::ExecuteTrace is used to clean up path-following issues, including handling edge cases and smart objects. The core of this call is as follows:

```
IPathFollower* pPathFollower = gAIEnv.CVars.PredictivePathFollowing ? pPipeUser-
>GetPathFollower() : 0;
bTraceFinished = pPathFollower ? ExecutePathFollower(pPipeUser, bFullUpdate,
pPathFollower) : Execute2D(pPipeUser, bFullUpdate);
```

COPTrace::Execute does the same work plus a bit more. For the AI following a path, when its lookahead position hits the end of the path, this operation sends the signal "OnEndWithinLookAheadDistance" to the AI. In the sample scenario, this allows our racing HMMWVs to start looking for a new path to follow while they're still moving along the current path. Normally AI agents stop moving when the path following process is completed. The following Lua script is also useful to maintain movement:

```
AI.SetContinuousMotion(vehicle.id, true);
```

COPTrace::Execute2D

This operation can be used as a fallback if an AI agent (CPipeUser, at least) doesn't have a path follower. COPTrace::Execute2D accomplishes the following tasks:

- Gets the lookahead path position and the path direction at this position.
- Executes a maneuver, if necessary. For example, it makes cars go backwards to make a U-turn.
- Considers a number of reasons to slow down, including:
  - The angle between current and desired (aforementioned path direction) directions.
  - The curvature of the path.
  - Approaching the end of the path.
  - Approaching the top of a hill.

It then sets members *fDesiredSpeed* and *vMoveDir* of the AI agent's SOBJECTSTATE structure, which are brought to the game code later. For an example of how this data can be used for actual steering, take a look at CVehicleMovementArcadeWheeled::ProcessAI.

Note that COPTrace::Execute2D is not the only operation that sets *vMoveDir*. For example, obstacle avoidance code can overwrite it.

Movement System

Key priorities for the AI Movement system include the following features.

- Robust and predictable. Navigation can be very unreliable, with no guarantee that a character will carry out the requested movement and end up at the desired destination. This is a very organic problem with no clear resolutions. The AI Movement system solves this by providing more explicit information about failure reasons.
- Central, clear ownership and easy debugging. Rather than having contextual movement information – style, destination, requester, etc. – tied to a specific goalop and getting lost when a behavior switch

occurs, Lumberyard maintains this information in a central location and separated from the goalop. In practice, a movement request can be sent from anywhere and the movement system handles it centrally. when the goalop requester is no longer interested, it simply cancels the request. This doesn't mean the character stops immediately and all information is lost, it just means that interest in the request has expired.

- Planning. In Lumberyard, logic is handled in blocks for ease of use and organization. Movement blocks are responsible for their own isolated tasks, such as FollowPath, LeaveCover and UseSmartObject. A collection of blocks in sequence make up a plan, which is produced by a controller with a string-pulled path as input. This types of organization helps clarify a larger picture about what is being processed right now and what is coming up.

Note
This system is still a work in progress, and it's design was focused on solving some critical problems with an existing code base. It may not be suitable for all game titles.

Using the Movement System

Using the movement system is pretty straightforward. Create a MovementRequest object with information about the destination, style and a callback. Queue it in MovementSystem and receive a MovementRequestID. Use this if you want to cancel the request. Then wait for MovementSystem to process to your request. Once your request is processed, you'll be notified via the callback.

Here's what's happening internally to process your request:

1. Once MovementSystem receives your request, it creates an internal representation of the character, called a MovementActor. This is a container for all internal states and the proxy to all external states/logic related to a character. It binds a MovementController to the actor. Currently there's only one controller available – GenericController, which is the result of what was done before. (The term "controller" is also used on the game side for a similar but different entity. These entities may be merged in the future, and multiple types of controllers added, such as for the Pinger, Scorcher, or BipedCoverUsed.)
2. MovementSystem informs the controller that there's a new request to start working on. GenericController kicks off the path finder.
3. Once the pathfinding result is in, the GenericController produces a plan that it starts to follow.
4. When the GenericController finishes the last block in the plan, it informs MovementSystem that the task is finished.
5. MovementSystem notifies the requester of success, and moves on to the next request.

Potential Improvements

The following areas of improvement or enhancement are under consideration:

- Change request processing. Currently there is a request queue, with movement requests processed one at a time, in FIFO order. Requests are immutable, so it's impossible to change a request once it's been queued; as a result, the only option is to cancel a request and queue a new one. These issues could be resolved by removing the request queue and allowing only one request at a time. If a request comes in while one is already being processed, interrupt the current one and report it.
- Validate a pipe user before proceeding with the update.
- When a UseSmartObject block detects that the exact positioning system fails to position a character at the start of a smart object, it reports this failure through the agent's bubble and in the log. It then resolves the problem by teleporting the character to the end of the smart object and proceeds to the next block in the plan.
- The GenericController is only allowed to start working on a new request while it is executing a FollowPath block. It then shaves off all subsequent blocks so that the actor doesn't find itself in the middle of a

smart object when planning takes place. This could be improved by allowing the controller to produce a part of the plan, looking further ahead, and then patch it with the current plan.

- The plan isn't removed when a request is canceled. This is because a subsequent 'stop' or 'move' request should follow the cancellation. However, until this request has been received, the controller has no way to know what to do.
- The pathfinding request is being channeled through the pipe user, and the result is returned to the pipe user as well as stored in `m_path`. This path is then extracted by the movement controller. It would be better if the pathfinder could be employed directly by the movement controller and skip the pipe user as a middle layer.
- The movement controller code would fit better on the game side, since that's where the information about the characters should live. It could be merged with the movement transitions that are handled on the game side.
- Being able to pull out a movement request at any time makes the code slightly more complex, because we can't rely on that fact that the controller is always working on a request that still exists. It may be better to keep the request, flag it as abandoned and clear the callback.
- The code could be improved by separating planning and plan execution into two different code paths instead of one.

Auto-Disable

You can save CPU time by not updating distant AI agents. Use the auto-disable feature to controlled updates either on a per-AI basis or globally.

Global auto-disable

- To control auto-disable for all vehicles: use the console variable `v_autoDisable`.
- To control auto-disable for all AI agents: use the console variable `ai_UpdateAllAlways`.

Per-AI auto-disable

Per-AI auto-disable is controlled by the entity property AutoDisable. Refer to the Lumberyard User Guide for more details on AI and vehicle entities. You can also change this property (and behavior) at run time.

- C++: `pAIActorProxy->UpdateMeAlways(true);`
- Lua: `AI.AutoDisable(entity.id, 1);`
- In Flow Graph Editor: turn **AI:AutoDisable** on or off for each AI.

AI Scripting

This collection of topics describes how to handle some key AI capabilities using scripting.

This section includes the following topics:

Communication System

AI communication is about playing sound/voice and/or animations at the right times in the course of the game.

Setting up communication for an AI agent requires the following steps:

- General set up:
  - Define communication channels. Channels are used to track the status of communication events for an AI.
  - Define communications. Communications detail specifically what activity should occur (and how) when the communication is called for. Communications are grouped into configurations.
  - Set up voice libraries. Voice libraries support localized dialogs, subtitles, and lip-syncing.
- Specify communication types for an AI using AI properties:
  - Point an AI's CommConfig property to a communication configuration, which contains the set of communications for that AI.
  - Point an AI's esVoice property to a voice library to use for that AI.
- Trigger a communication event:
  - Specify the name of a communication channel for the event.
  - Specify the name of a communication to fire.

Communications, channels, and voice libraries are defined in a set of XML files. At game start-up, the folder `Game/Scripts/AI/Communication` and all subfolders are scanned for XML files containing these configurations.

Defining Communication Channels

A communication channel determines whether an AI can play a communication at a given moment, depending on whether or not the communication channel is occupied. Channels are a self-contained concept, independent of other AI communication concepts. They have a sole purpose: to be in one of two possible states, "occupied" or "free".

AI communication channels are defined in an XML file stored in `Game/Scripts/AI/Communication`. The SDK includes a template channel configuration XML file, called `ChannelConfg.xml`. Communication channels are configured in a hierarchy of parent and child channels. The hierarchical structure determines how a channel's occupied status affects the status of other channels (for example, a parent of an occupied child channel).

Channel Elements & Attributes

Communication channels are defined in a `<ChannelConfig>` element with the following attributes:

name
Channel name.

priority

minSilence
Minimum time (in seconds) that the channel should remain occupied after a communication has been completed.

flushSilence
Time (in seconds) that the channel should remain occupied after it has been flushed. This value overrides the imposed silence time (**minSilence**) after playing a communication. If not specified, the value set for **minSilence** is used.

actorMinSilence
: Minimum time (in seconds) to restrict AI agents from playing voice libraries after starting a communication.

ignoreActorSilence
: Flag indicating that AI agent communication restrictions from the script should be ignored.

type
: Type of communication channel. Valid values are "personal", "group" or "global".

Example

```
Game/Scripts/AI/Communication/ChannelConfig.xml

<Communications>
    <ChannelConfig>
       <Channel name="Global" minSilence="1.5" flushSilence="0.5" type="global">

            <Channel name="Group" minSilence="1.5" flushSilence="0.5"
type="group">
                <Channel name="Search" minSilence="6.5" type="group"/>
                <Channel name="Reaction" priority="2" minSilence="2" flush
Silence="0.5" type="group"/>
                <Channel name="Threat" priority="4" minSilence="0.5" flush
Silence="0.5" type="group"/>
            </Channel>
            <Channel name="Personal" priority="1" minSilence="2" actorMinSi
lence="3" type="personal"/>
        </Channel>
    </ChannelConfig>
</Communications>
```

Configuring Communications for an AI

Communication configurations determine what communication activity AI agents can perform and how it will manifest. Communications for a particular type of AI are grouped into configurations. For example, your game might have both human and non-human AI agents, each with its own set of communication activities. In this scenario, you might group all the human communications into a configuration object named "human" while communications for non-humans might be grouped into a "non-human" configuration. For a particular AI, you'll specify the configuration to use with the AI's *CommConfig* property. With this configuration structure, you can define a communication (such as "surprise") differently in each configuration so that, when triggered, the communication activity fits the AI involved.

For each communication, you also have the option to define multiple variations of action and specify how the variations are used.

AI communication channels are defined in one or more XML files stored in `Game/Scripts/AI/Communication`. The SDK includes a template channel configuration XML file, called `BasicCommunications.xml`.

Communication Elements & Attributes

Communications are configured using the following elements and attributes:

Config

Communication configurations are grouped into `<Config>` elements and use the following attributes. Each configuration must contain at least one communication.

name
Configuration name, which can be referenced in the AI's CommConfig property.

Communication

A communication is defined in a `<Communication>` element with the following attributes. Each communication should contain at least one variation.

name
Communication name.

choiceMethod
Method to use when choosing a variation. Valid values include "Random", "Sequence", "RandomSequence" or "Match" (uses only the first variation).

responseName

responseChoiceMethod
Similar to **choiceMethod**.

forceAnimation
Boolean flag.

Variation

Each variation is defined in a `<Variation>` element with the following attributes.

animationName
Animation graph input value.

soundName

voiceName

lookAtTarget
Boolean flag indicating whether or not the AI should look at the target during the communication.

finishMethod
Method used to determine when the communication is finished, such as after the communication type has finished or after a time interval. Valid values include "animation", "sound", "voice", "timeout" or "all".

blocking
AI actions that should be disabled during the communication. Valid values include "movement", "fire", "all", or "none".

animationType
Valid values include "signal" or "action".

timeout

Example

```
Game/Scripts/AI/Communication/BasicCommunications.xml

<Communications>
<!--sound event example-->
    <Config name="Welcome">
       <Communication name="comm_welcome" finishMethod="sound" blocking="none">
```

```
            <Variation soundName="sounds/dialog:dialog:welcome" />
        </Communication>
    </Config>
<!--example showing combined animation + sound event (needs state using ac
tion/signal in the animation graph)-->
    <Config name="Surprise">
       <Communication name="comm_anim" finishMethod="animation" blocking="all"
 forceAnimation="1">
          <Variation animationName="Surprise" soundName="sounds/interface:play
er:heartbeat" />
        </Communication>
    </Config>
</Communications>
```

Setting Up Voice Libraries

To support localized dialogs, subtitles, and lip syncing, you need to set up voice libraries. Once set up, you can assign a voice library to an AI (or entity archetype) using the AI's *esVoice* property.

Voice libraries are defined in a set of XML Excel files stored in `GameSDK/Libs/Communication/Voice`. The SDK includes a template voice library file at `GameSDK/Libs/Communication/Voice/npc_01_example.xml`.

Each voice library must include the following information.

Language
Localization type for this library.

File Path
Location where the sound files for this library are stored.

Signal
Communication name associated with a sound file.

Sound File
File name of a sound file, listed by signal.

Example
Comment field used to describe or illustrate a sound file.

Example

GameSDK/Libs/Communication/Voice/npc_01_example.xml

Language	American English	
File Path	languages/dialog/ai_npc_01/	
Signal	**Sound File**	**SDK NPC 01 Example**
see		
	see_player_00	i see you
	see_player_01	hey there you are
	see_player_02	hey i have been looking for you
pain		

	pain_01	ouch
	pain_02	ouch
	pain_03	ouch
death		
	death_01	arrrhh
	death_02	arrrhh
	death_03	arrrhh
alerted		
	alerted_00	watch_out
	alerted_01	be careful
	alerted_02	something there

Setting Communication for an AI

An AI's communication methods are set using the AI agents properties. You can set AI properties in several ways. For information about using the Lumberyard Editor to set AI properties, see "Using Database View to Set AI Communication" in the Lumberyard User Guide).

Set the following properties:

- *CommConfig* – Set this property to the name of the communication configuration you want this AI to use. Communication configurations are defined in XML files in `Game/Scripts/AI/Communication`, using `<Config>` elements.
- *esVoice* – Set this property to the name of the XML file containing the voice library you want this AI to use. Voice libraries are defined in XML files in `GameSDK/Libs/Communication/Voice`.

Turning Animation and Voice Off

Communication animation and/or voice can be turned off for an AI agent using the agent's Lua script (as in the example below) or the entity properties in Lumberyard Editor Editor.

Example

```
Game/Scripts/Entities/AI/Shared/BasicAITable.lua

Readability =
{
   bIgnoreAnimations = 0,
   bIgnoreVoice = 0,
},
```

Triggering a Communication Event

To trigger a communication event, use the goalop *communicate* with the following attributes. Note that communication animations are not played if the AI is currently playing a smart object action.

name
: Name of the communication to trigger (sound, voice, and/or animation). Communication names are defined in an XML file referred to by the CommConfig property of this AI.

channel
: Communication channel being used by this AI. An AI's communication channel is defined in an XML file in `Game/Scripts/AI/Communication`.

expirity (expiry)
: Maximum allowable delay in triggering the communication when the communication channel is temporarily occupied. If a communication can't be triggered within this time period, it is discarded.

To trigger communications using flow graph logic, use the Flow Graph node **AI:Communication**.

Example

```
<GoalPipe name="Cover2_Communicate">
   <Communicate name="comm_welcome" channel="Search" expirity="0.5"/>
</GoalPipe>
```

Debugging

To get debug information on AI communication issues, use the following console variables (`ai_DebugDraw` should be set to "1"):

- ai_DebugDrawCommunication
- ai_DebugDrawCommunicationHistoryDepth
- ai_RecordCommunicationStats

Debug output is shown in the console as illustrated here:

```
Playing communication: comm_welcome[3007966447] as playID[84]
CommunicationPlayer::PlayState: All finished! commID[-1287000849]
CommunicationManager::OnCommunicationFinished: comm_welcome[3007966447] as
playID[84]
CommunicationPlayer removed finished: comm_welcome[3007966447] as playID[84]
with listener[20788600]
```

Troubleshooting

[Warning] Communicate(77) [Friendly.Norm_Rifle1] Communication failed to start

You may get this message or a similar one if your AI's behavior tree calls a communication but the communication configuration is not set up properly. In this example message, "77" refers to line 77 in your AI's behavior tree script (or goalop script). This line is probably communication trigger such as this:

```
<Communicate name="TargetSpottedWhileSearching" channel="Reaction" expirity="1.0"
 waitUntilFinished="0" />
```

Some things to check for::

- Does the specified communication name "TargetSpottedWhileSearching" exist in your communication configuration files (XML files located in `Game/Scripts/AI/Communication/`)?

- Check the *CommConfig* property for the AI. Is it set to the name of a `<Config>` element defined in your communication configuration files? If so, is the communication name "TargetSpottedWhileSearching" defined inside this `<Config>` element? This issue, calling communications that aren't configured for the AI is a common source of this error.
- Check the communication's variation definition. Does it point to a resource (animation, sound) that exists? If using a voice library, does it point to a valid voice library file name?

Factions

AI agents use factions to determine their behavior when encountering other AI agents. There are a base set of behaviors such as neutral, friendly and hostile. For example, when an AI in the "Grunt" faction encounters an AI in the "Players" faction, the encounter will be hostile. Players encountering "Civilians" will be friendly, etc.

To set up faction communications:

- Create an XML file that defines all the factions in your game and their reactions to each other (see the example). This file should be placed in `\Games\Scripts\AI\`. The SDK includes a template faction XML file, called `Factions.xml`.
- Set the Faction property for all of your AI agents to one of the defined factions. You can also set factions using Flow Graph

Example: Faction setup

```
Factions.xml

<Factions>
    <Faction name="Players">
        <Reaction faction="Grunts" reaction="hostile" />
        <Reaction faction="Civilians" reaction="friendly" />
        <Reaction faction="Assassins" reaction="hostile" />
    </Faction>
    <Faction name="Grunts">
        <Reaction faction="Players" reaction="hostile" />
        <Reaction faction="Civilians" reaction="neutral" />
        <Reaction faction="Assassins" reaction="hostile" />
    </Faction>
    <Faction name="Assassins">
        <Reaction faction="Players" reaction="hostile" />
        <Reaction faction="Civilians" reaction="hostile" />
        <Reaction faction="Grunts" reaction="hostile" />
    </Faction>
    <Faction name="HostileOnlyWithPlayers" default="neutral">
        <Reaction faction="Players" reaction="hostile" />
    </Faction>
    <Faction name="Civilians" default="neutral" />
    <Faction name="WildLife" default="neutral" />
</Factions>
```

Modular Behavior Tree

Modular behavior tree (MBT) is a collection of concepts for authoring behaviors for artificial intelligent (AI) agents in your game. Instead of writing complicated code in C++ or other general purpose programming language, MBT lets you describe AI behaviors at a high level without having to think about mechanics such as pointers, memory, and compilers. MBT concepts and implementation are optimized for rapid iteration and re-use.

Core Concepts

Conceptually, MBT is based on two key objects: the *node* and the *tree*.

Node

The node is the most fundamental concept; it is a building block that can be combined with others to build behaviors. A node consists of a block of code that represents a simple task. All nodes have the same interface: when processed, they carry out a task and either succeed or fail.

Nodes can be stand-alone or may have child nodes, which are processed as part of the parent node processing. When processed, the success of a parent node often (but not always) depends on the success of each child node.

Nodes follow several common patterns, such as action, composite, and decorator nodes. These common node patterns are more fully described in later in this topic.

Game developers can create the nodes needed for their game. In addition, Lumberyard provides a set of standard nodes for general use. These include nodes for tasks related to AI, animation, flying, and common game activities, as well as generic nodes useful when building behaviors, such as for timeouts and looping tasks. These provided nodes are documented in the Modular Behavior Tree Node Reference (p. 52).

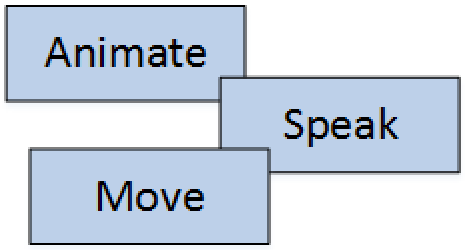
Tree

Behaviors are constructed by building trees of nodes, collections of individual tasks that, when positioned as a root with branches that extend out into leaves, define how an AI agent will behave in response to input.

Common Node Patterns

Action Nodes

An action node represents some sort of simple action. Action nodes might cause the AI agent to speak,



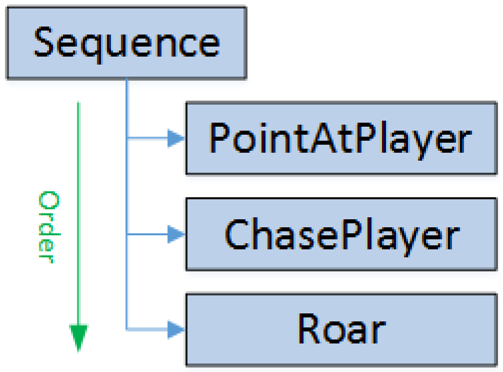
play an animation, or move to a different location.

Composite Nodes

A composite node represents a series of actions to be performed in a certain order. Composite nodes consist of a parent node and two or more child nodes. Whether or not a child node is processed (and in what order) can depend on the success or failure of previously processed nodes. Common composite patterns include sequential, selector, and parallel.

Sequential node

This composite pattern describes child nodes that are processed consecutively in a specified sequence. All child nodes are processed regardless of whether the previous child node succeeded or failed. For example, a sequential node might cause an AI monster to point at the player, roar, and then run toward the player. In this pattern, each child node in the sequence must succeed for the next child node to start processing; if any child node fails, the parent node immediately fails and processing is

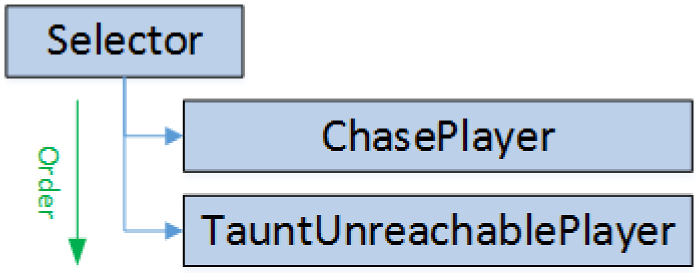


stopped.

Selector node

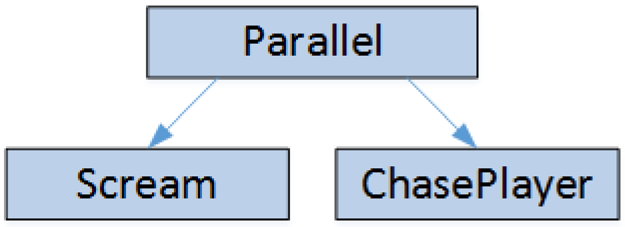
This composite pattern describes child nodes that are processed consecutively and in sequence only until one succeeds. As soon as one child node succeeds, the parent node succeeds immediately and stops processing child nodes. If all child nodes are attempted and all fail, the parent node fails. This pattern is useful for setting up AI agents to try multiple different tactics, or for creating fallback behaviors to handle unexpected outcomes.

Imagine, for example, that we want our AI monster to chase the player, but if it can't reach the player it should scream “Come and fight me, you coward!” To implement this scenario, a selector parent node is set up with two children, one for each possible action. The parent node first processes the “chase player” child node. If it succeeds, then the selector node stops there. However, if the “chase player node fails, then the parent node continues and processes the “taunt player” child node.



Parallel node

This composite pattern describes child nodes that are processed concurrently. In this scenario, Imagine we want our AI monster to scream and chase the player at the same time rather than one



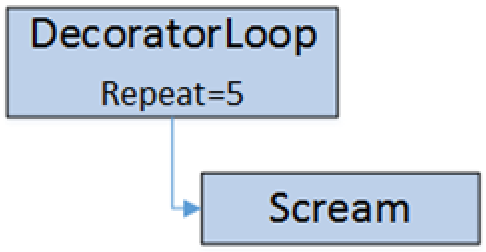
after the other.

Decorator Nodes

A decorator node represents some sort of functionality that can be added to another node and behaves regardless of how the other node works or what it does. Common decorator functionality includes looping and limiting concurrent functionality.

Looping

Looping functionality can be used to process any other node multiple times. Rather than creating custom nodes every time you want to repeat a task, you can wrap any node in a parent loop decorator node. By setting a parameter for the loop node, you can dictate the number of times the child nodes will be processed. Each time the child node succeeds, the loop node count is updated and the child node is re-processed. Once the loop count meets the set parameter, the loop node succeeds.

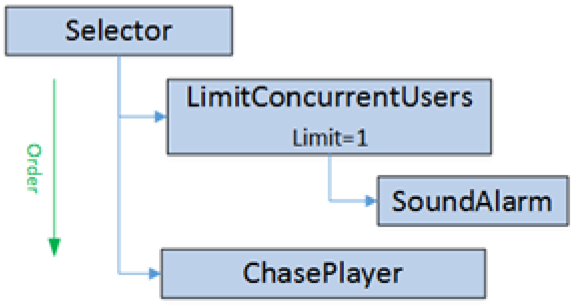


Limiting concurrent users

This functionality lets you specify how many users should be allowed to concurrently use a specified node. It is a good way to ensure variations in behavior among a group of AI agents. A typical scenario illustrating this function is as follows: The player is spotted by a group of three monsters. You want one monster to sound an alarm while the others chase the player.

Limiting concurrent users works with a selector node, which steps through a sequence of child nodes until one succeeds. By wrapping one of a selector node's child nodes in a limit decorator node, you can cause the child node to fail due to concurrent users, which in turn causes the selector node to move to the next child.

To handle the scenario described, the selector node would have two child nodes, “sound alarm” and “chase player”. The “sound alarm” node is wrapped in a limit node, with the user limit set to 1. Monster #1 flows through the selector node to the limit node; as there is no one currently using the “sound alarm” node, the Monster #1 takes this action. The limit node records that one AI agent is processing the child node, so effectively locks the door to it. Monsters #2 and #3 also flow through the selector node to the limit node, but because the limit node has reached its limit of user, it reports a failure. Consequently, the selector node moves on to the next child node in the sequence, which is “chase



player”. So monsters #2 and #3 chase the player.

Describing Behavior Trees in XML

Behavior trees are described using XML markup language. Behavior trees are hot-loaded every time the user jumps into the game in the editor.

The following XML example describes the behavior tree for a group of monsters. In this example, only one monster at a time is allowed to chase the player. The remaining monsters stand around and taunt the player.

```
<BehaviorTree>
    <Root>
        <Selector>
            <LimitConcurrentUsers max=”1”>
                <ChasePlayer />
            </LimitConcurrentUsers>
            <TauntPlayer />
        </Selector>
    </Root>
</BehaviorTree>
```

C++ Implementation

You'll find all MBT code encapsulated in the BehaviorTree namespace.

Understanding the Memory Model

MBT has a relatively small memory footprint. It accomplishes this by (1) sharing immutable (read-only) data between instances of a tree, and (2) only allocating memory for things that are necessary to the current situation.

Memory is divided into two categories: configuration data and runtime data. In addition, MBT uses smart pointers.

Configuration data

When a behavior tree such as the following example is loaded, a behavior tree template is created that holds all the configuration data shown in the example. This includes a sequence node with four children: two communicate nodes, an animate node, and a wait node. The configuration data is the animation name, duration, etc., and this data never changes.

```
<Sequence>
    <Communicate name=”Hello” />
    <Animate name=”LookAround” />
    <Wait duration=”2.0” />
    <Communicate name=”WeShouldGetSomeFood” />
</Sequence>
```

Memory for the configuration data is allocated from the level heap. When running the game through the launcher, this memory is freed on level unload; alternatively, it is freed when the player exits game mode and returns to edit mode in Lumberyard Editor.

Runtime data

When spawning an AI agent using a behavior tree, a behavior tree Instance is created and associated with the agent. The instance points to the behavior tree template for the standard configuration data, which means that the instance contains only instance-specific data such as variables and timestamps.

When the tree instance is accessed for the AI agent, it begins by executing the Sequence node. If the core system detects that this is the first time the behavior has been run for this AI agent, it allocates a runtime data object specifically for this node and agent. This means that every AI agent gets its own runtime data object when executing a behavior tree node. The runtime data object persists as long as the AI agent is executing a node (this can be several frames) but is freed when the AI agent leaves a node.

Memory for runtime data is allocated from a bucket allocator. This design minimizes memory fragmentation, which is caused by the fact that runtime data is usually just a few bytes and is frequently allocated and freed. The bucket allocator is cleaned up on level unload.

Smart pointers

MBT uses Boost smart pointers to pass around data safely and avoid raw pointers as much as possible. Memory management is taken care of by the core system. (While there are circumstances in which a *unique_ptr* from C++11 would work well, Lumberyard uses Boost's *shared_ptr* for compatibility reasons.)

Implementing an MBT Node

To implement a new MBT node in C++, you'll need to do the following tasks:

- Create the node
- Expose the node to the node factory
- Set up error reporting for the node

Creating a node

The following code example illustrates a programmatic way to create a behavior tree node. When naming new nodes, refer to Recommended Naming Practices (p. 51).

```
#include <BehaviorTree/Node.h>

class MyNode : public BehaviorTree::Node
{
    typedef BehaviorTree::Node BaseClass;

    public:
    // Every instance of a node in a tree for an AI agent will have a
    // runtime data object. This data persists from when the node
    // is visited until it is left.
    //
    // If this struct is left out, the code won't compile.
    // This would contain variables like 'bestPostureID', 'shotsFired' etc.
    struct RuntimeData
    {
    };

    MyNode() : m_speed(0.0f)
    {
    }

    // This is where you'll load the configuration data from the XML file
   // into members of the node. They can only be written to during the loading
 phase
    // and are conceptually immutable (read-only) once the game is running.
    virtual LoadResult LoadFromXml(const XmlNodeRef& xml, const LoadContext&
context)
    {
        if (BaseClass::LoadFromXml(xml, context) == LoadFailure)
            return LoadFailure;
        xml->getAttr("speed", m_speed);
        return LoadSuccess;
    }
protected:
    // Called right before the first update
    virtual void OnInitialize(const UpdateContext& context)
    {
        BaseClass::OnInitialize(context);

        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

    // Called when the node is terminated
    virtual void OnTerminate(const UpdateContext& context)
    {
        BaseClass::OnTerminate(context);

        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

    virtual Status Update(const UpdateContext& context)
    {
        // Perform your update code and report back whether the
        // node succeeded, failed or is running and needs more
        // time to carry out its task.

        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
        return Success;
    }

    // Handle any incoming events sent to this node
    virtual void HandleEvent(const EventContext& context, const Event& event)
    {
        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

private:
    // Store any configuration data for the node right here.
    // This would be immutable things like 'maxSpeed', 'duration',
    // 'threshold', 'impulsePower', 'soundName', etc.
    float m_speed;
};

// Generate an object specialized to create a node of your type upon
// request by the node factory. The macro drops a global variable here.
GenerateBehaviorTreeNodeCreator(MyNode);
```

Exposing a node

To use the newly created node, you'll need to expose it to the node factory, as shown in the following code snippet.

```
BehaviorTree::INodeFactory& factory = gEnv->pAISystem->GetIBehaviorTreeManager()-
>GetNodeFactory();
ExposeBehaviorTreeNodeToFactory(factory, MyNode);
```

Setting up error reporting

Use the class `ErrorReporter` to report errors and warnings in the new node. It will let you log a printf-formatted message and automatically include any available information about the node, such as XML line number, tree name, and node type.

```
ErrorReporter(*this, context).LogError("Failed to compile Lua code '%s'",
code.c_str());
```

Variables

Variables are statically declared in XML, with information about how they will change in response to signals from AI agents (named text messages within the AI system).

The following code snippet illustrates the use of variables to receive input from the AI system. In this example, the AI agent takes action based on whether or not it can "see" the target.

```
<BehaviorTree>
    <Variables>
        <Variable name="TargetVisible" />
    </Variables>
    <SignalVariables>
        <Signal name="OnEnemySeen" variable="TargetVisible" value="true" />
       <Signal name="OnLostSightOfTarget" variable="TargetVisible" value="false"
 />
    </SignalVariables>
    <Root>
        <Selector>
            <IfCondition condition=”TargetVisible”>
                <Move to=”Target” />
            </IfCondition>
            <Animate name=”LookAroundForTarget” />
        </Selector>
    </Root>
</BehaviorTree>
```

Lua Scripting

Lua code can be embedded in a behavior tree and executed along with the tree nodes. This is useful for running fire-and-forget code or for controlling the flow in a tree. It's useful for prototyping or extending functionality without having to create new nodes.

The code is compiled once when the level is loaded in pure game to reduce fragmentation. Only code for behavior trees that are actually used in that level will be compiled.

All Lua nodes provide access to the *entity* variable.

- `ExecuteLua` runs a bit of Lua code. It always succeeds.

  ```
  <ExecuteLua code=”DoSomething()” />
  ```

- `LuaWrapper` inserts a bit of Lua code before and after running child node. The post-node code is run regardless of whether the child node succeeded or failed.

```
<LuaWrapper onEnter=”StartParticleEffect()” onExit=”StopParticleEffect()”>
    <Move to=”Cover” />
</LuaWrapper>
```

- `LuaGate` uses a bit of Lua code to control whether or not a child node should be run. If the Lua code returns true, the child node is run and `LuaGate` returns the status of the child node (success or failure). If the code returns false or fails to execute, the child node is not run, and `LuaGate` returns failure.

```
<LuaGate code=”return IsAppleGreen()”>
    <EatApple />
</LuaGate>
```

- `AssertLua` lets you make a statement. If the statement is true, the node succeeds; if it's false the node fails.

```
<Sequence>
    <AssertLua code=”return entity.someCounter == 75” />
    <AssertCondition condition=”TargetVisible” />
    <Move to=”Target” />
</Sequence>
```

Timestamps

A timestamp identifies a point in time when an event happened. A lot of AI behavior depends on tracking the timestamp of certain events and measuring the amount of time from those points. For example, it can be useful to tie behavior to how long it's been since the AI agent was last shot at or hit, when it last saw the player, or how long it's been since moving to the current cover location.

Timestamps can be declared as mutually exclusive, that is, both timestamps can't have a value at the same time. For instance, `TargetSpotted` and `TargetLost` can both have a value because the AI agent can't see a player and at the same time consider them lost. With exclusive timestamps, when one timestamp has a value written to it, the other timestamp is automatically cleared.

The following code snippet illustrates the use of timestamps.

```
<BehaviorTree>
    <Timestamps>
        <Timestamp name="TargetSpotted" setOnEvent="OnEnemySeen" />
        <Timestamp name="ReceivedDamage" setOnEvent="OnEnemyDamage" />
        <Timestamp name="GroupMemberDied" setOnEvent="GroupMemberDied" />
    </Timestamps>
    <Root>
        <Sequence>
            <WaitUntilTime since=”ReceivedDamage” isMoreThan=”5” orNever
BeenSet=”1” />
            <Selector>
                <IfTime since="GroupMemberDied" isLessThan=”10”>
                    <MoveCautiouslyTowardsTarget />
                </IfTime>
                <MoveConfidentallyTowardsTarget />
            </Selector>
        </Sequence>
    </Root>
</BehaviorTree>
```

Events

Communication with AI agents is done using AI signals, which essentially are named text messages. Signals such as OnBulletRain and OnEnemySeen communicate a particular event, which, when broadcast to other AI agents, can be reacted to based on each AI agent's behavior tree. This design allows AI behavior to remain only loose coupled with AI signals. AI Signals are picked up and converted to MBT events, then dispatched to the root node, which passes them along down the running nodes in the tree.

```
<Sequence>
    <WaitForEvent name=”OnEnemySeen” />
    <Communicate name=”ThereHeIs” />
</Sequence>
```

Debugging and Tree Visualization

This section provides help with debugging behavior trees by providing a tree visualization view during debugging. This view allows you to track an AI agent's progress through the tree as the game progresses.

"Slashing" Agents

This feature allows you to view the behavior tree for a specific AI agent in DebugDraw. To enable this feature:

1. Set ai_DebugDraw to 0 or 1 (default is -1).
2. Select the AI agent you want to view a behavior tree for:
   - Place the selected AI agent in the center of the camera view and press the numpad "/" key.
   - Call "ai_DebugAgent closest" to select the agent closest to the camera.
   - Call "ai_DebugAgent centerview" to select the agent closest to the center of the camera view (same as slash).
   - Call "ai_DebugAgent <AgentName>" to select a specific agent by its name.
   - Call "ai_DebugAgent" without a parameter to remove the tree visualization.

The tree visualization displays the AI agent's name at the top of the screen and identifies the agent on the screen with a small green dot. Tree nodes are displayed and color coded as follows, with line numbers from the XML file shown on the left.

- White – nodes with custom data
- Blue – leaf nodes, which often carry special weight when debugging
- Gray – all other nodes

Adding Custom Debug Text

Tree visualization supports custom node information. This allows you to get a more in-depth view of the currently running parts of a behavior tree. For example, you can see the name of the event that the `WaitForEvent` node is waiting for, or how much longer `Timeout` is going to run before it times out.

To use this feature, override `GetDebugTextForVisualizer`, as follows.

```
#ifdef STORE_INFORMATION_FOR_BEHAVIOR_TREE_VISUALIZER
virtual void GetDebugTextForVisualizer(
        const UpdateContext& updateContext,
        stack_string& debugText) const
{
    debugText.Format("Speed %f", m_speed);
}
#endif
```

Logging and Tracing

Tracing log messages is a critical tool for diagnosing problems. Lumberyard provides native support for logging, as shown in the following code snippet.

```
<Sequence>
    <QueryTPS name="CoverFromTarget" _startLog="Finding cover" _fail
ureLog="Failed to find cover" />
    <Move to="Cover" _startLog="Advancing" _failureLog="Failed to advance"
_successLog="Advanced" />
</Sequence>
```

(The reserved attributes _startLog, _successLog, and _failureLog are automatically read in.)

Log messages are routed through an object deriving from the `BehaviorTree::ILogRouter` interface. This allows you to determine where the logging messages end up. For example, one option would be to route the info to a personal log and store a short history of log messages for each AI agent; with this approach, log messages can be displayed when debugging as part of an AI agent's tree visualization.

The AI Recorder also retains all log messages; use this tool to explore sequences of events.

Compiling with Debug Information

To compile a game with debug information, you need to define DEBUG_MODULAR_BEHAVIOR_TREE.

```
#if !defined(_RELEASE) && (defined(WIN32) || defined(WIN64))
# define DEBUG_MODULAR_BEHAVIOR_TREE
#endif
```

Viewing Completed Trees

When a behavior tree finishes executing—either by failing or succeeding all the way through the root node, a notification is displayed in the console window along with a list of recently visited nodes and their line numbers.

[Error] Modular Behavior Tree: The root node for entity 'HumanSoldier' FAILED. Rebooting the tree next frame. (124) Move. (122) Selector. (121) Sequence.

Note that in the example above the tree will be rebooted in the next frame. This suggests that the behavior tree was not designed to handle a failure at this point.

Recommended Naming Practices

The following suggestions help streamline code clarity and communication in a development team.

Naming Nodes

For action nodes, use names that identify the action the node will perform. These are usually action verbs.

Good

- Loop
- Animate
- LimitConcurrentUsers
- ExecuteLua
- Shoot
- AdjustCoverStance

Bad

- Fast
- PathPredictor
- Banana
- Script
- ActivationProcess

Naming Timestamps

Name timestamps based on the event they're related to. Because timestamps describe an event that has already happened, use the past tense (TargetSpotted, not TargetSpots).

- TargetSpotted
- ReceivedDamage
- GroupMemberDied

Modular Behavior Tree Node Reference

This section contains reference information on modular behavior tree (MBT) node types. MBT node types are organized here based on the system they are defined into.

It is possible to expose MBT nodes from anywhere in Lumberyard code. A node can have parameters that configure the behavior of its execution. If an invalid value is passed to the node, causing the node's parsing to fail, an error message is written to either `Editor.log` or `Game.log`.

Node Index

Generic Nodes

These nodes provide the basic functionality of MBT.

Loop

Executes a single child node a specified number of times or until the child fails its execution.

Parameters

count
Maximum number of times the child node will be executed. If left blank, it is assumed to be infinite and the node will continue running until failure.

Success/Failure

The node SUCCEEDS if the maximum number of repetitions is reached. The node FAILS if execution of the child node FAILS.

Example

```
<Loop count="3">
    <SomeChildNode />
</Loop>
```

LoopUntilSuccess

Executes a child node a specified number of times or until the child node succeeds its execution.

Parameters

attemptCount
Maximum number of times the child node will be executed. If left blank or set to <=0, it is assumed to be infinite and the node will continue running until success.

Success/Failure

The node SUCCEEDS if the child SUCCEEDS. The node FAILS if the maximum amount of allowed attempts is reached.

Example

```
<LoopUntilSuccess attemptCount="5">
    <SomeChildNode />
</LoopUntilSuccess>
```

Parallel

Executes its child nodes in parallel.

Note

- A maximum number of 32 child nodes is allowed.
- When success and failure limits are reached at the same time, the node will succeed.

Parameters

failureMode
Method to use to evaluate when the node fails. Acceptable values include "any" or "all". Default: "any".

successMode
Method to use to evaluate when the node succeeds. Acceptable values include "any" or "all". Default: "all".

Success/Failure

When `successMode` is set to "all", the node SUCCEEDS if all the child nodes SUCCEEDS.

When `successMode` is set to "any", the node SUCCEEDS if any of the child nodes SUCCEED.

When `failureMode` is set to "any", the node FAILS if any of the child nodes FAILS.

When `failureMode` is set to "all", the node FAILS if all of the child nodes FAIL.

Example

```
<Parallel successMode="any" failureMode="all">
    <SomeChildNode1 />
    <SomeChildNode2 />
    <SomeChildNode3 />
</Parallel>
```

Selector

Executes its child nodes consecutively, one at a time, stopping at the first one that succeeds.

Parameters

None.

Success/Failure

The node executes the child nodes in sequential order and SUCCEEDS as soon as one of the child SUCCEEDS. Once the node succeeds, the child nodes that follow are not executed. The node FAILS if all the child nodes FAIL.

Example

```
<Selector>
    <SomeChildNode1 />
    <SomeChildNode2ToExecuteIfSomeChildNode1Fails />
    <SomeChildNode3ToExecuteIfSomeChildNode2Fails />
</Selector>
```

Sequence

Executes its child nodes one at a time in order.

> **Note**
> A maximum of 255 child nodes is allowed.

Parameters

None.

Success/Failure

The node SUCCEEDS if all the child nodes SUCCEED. The node FAILS if any of the child nodes FAILS.

Example

```
<Sequence>
    <SomeChildNode1 />
    <SomeChildNode2 />
    <SomeChildNode3 />
</Sequence>
```

StateMachine

Executes child nodes of type `State` one at a time. The first child node defined is the first to be executed. The current status of a StateMachine node is the same as that of the child that is currently being executed.

Parameters

None.

Success/Failure

The node SUCCEEDS if the current State child node SUCCEEDS. The node FAILS if the current State child node FAILS.

Example

```
<StateMachine>
    <State />
    <State name="State1" />
    <State name="State2" />
</StateMachine>
```

State & Transitions

Executes the content of its BehaviorTree node. This node can transition to another state (or itself). If a State node is instructed to transition into itself while running, it will first be terminated, re-initialized, and then updated again.

A State node has the following characteristics:

- Is a basic block of a StateMachine node.
- MUST have a BehaviorTree node.
- MAY have a Transitions element.

Transitions

Transitions elements are described inside a State node, and can contain the definitions of as many transitions as are needed. The transitions elements are not MBT nodes. If a transition specifies a destination state that doesn't exist, an error message will be displayed when parsing the MBT node.

Parameters

`<State />` elements must include the following parameters:

name
Name of the state. It must be unique within the scope of the StateMachine it is in.

`<Transition />` elements must include the following parameters:

onEvent
Name of the event that may cause the transition to happen. These events are of type AISignal.

to
Name of the state to transition to.

Success/Failure

The node SUCCEEDS if the content of the BehaviorTree node SUCCEEDS.

The node FAILS if the content of the BehaviorTree node FAILS.

Example

```
<State name="StateName">
 <Transitions>
  <Transition onEvent="EventOrTransitionSignalName" to="OtherStateName" />
 </Transitions>
 <BehaviorTree>
  <SomeChildNode />
 </BehaviorTree>
</State>
```

SuppressFailure

Owns and executes one child node. This node will succeed regardless of whether the child node succeeds.

Parameters

None.

Success/Failure

The node always SUCCEEDS once the child node has been executed.

Example

```
<SuppressFailure>
    <SomeChildThatCanFail />
</SuppressFailure>
```

Timeout

Fails once a certain amount of time has passed.

Parameters

duration
Amount of time (in seconds) before failure occurs.

Success/Failure

The node FAILS if it runs for more than the amount of time specified in the `duration` parameter.

Example

```
<Timeout duration=5" />
```

Wait

Succeeds once a certain amount of time has passed.

Parameters

duration
Amount of time (in seconds) before the request succeeds.

variation
Maximum additional amount of time that may be randomly added to the value of `duration`, in the range [0, `variation`]. Setting this value causes the wait time to have random variations between different executions of the node.

Success/Failure

The node SUCCEEDS once it has run the duration specified (plus random variation).

Example

```
<Wait duration="5" variation="1" />
```

AI Nodes

These nodes provide MBT functionality for the AI system.

AdjustCoverStance

Updates the AI agent's cover stance based on the maximum height at which its current cover is effective.

Parameters

duration
(Optional) Length of time (in seconds) the node will execute. Set to **`continuous`** to specify an unlimited time span.

variation
(Optional) Maximum additional time (in seconds) that may be randomly added to the value of `duration`, in the range [0, `variation`]. Setting this value causes the wait time to have random variations between different executions of the node.

Success/Failure

The node SUCCEEDS if execution of the child runs the length of the specified duration. The node FAILS if the child is not in cover.

Example

```
<AdjustCoverStance duration="5.0" variation="1.0"/>
```

Aim

Sets a location for the AI agent to aim at, and then clears the location when the node stops executing.

Parameters

at
Location to aim at. Allowed values include:

- RefPoint
- Target

angleThreshold
(Optional) Tolerance angle for aim accuracy.

durationOnceWithinThreshold
(Optional) Amount of time (in seconds) to continue aiming.

Success/Failure

The node SUCCEEDS after aiming at the desired location for the specified duration, if the location is not valid or if the timeout elapses.

Example

```
<Aim at="Target" durationOnceWithinThreshold="2.0" />
```

AimAroundWhileUsingAMachingGun

Updates the aim direction of the AI agent when using a mounted machine gun.

Parameters

maxAngleRange
(Optional) Maximum angle to deviate from the original direction.

minSecondsBeweenUpdates
(Optional) Minimum amount of delay (in seconds) between updates.

useReferencePointForInitialDirectionAndPivotPosition
Boolean.

Success/Failure

The node does not succeed or fail.

Example

```
<AimAroundWhileUsingAMachingGun minSecondsBeweenUpdates="2.5" maxAngleRange="30"
 useReferencePointForInitialDirectionAndPivotPosition="1"/>
```

Animate

Sets the AI agent to play an animation.

Parameters

name
Animation to be played.

urgent
(Optional) Boolean indicating whether or not to add the urgent flag to the animation.

loop
(Optional) Boolean indicating whether or not to add the loop flag to the animation.

setBodyDirectionTowardsAttentionTarget
(Optional) Boolean indicating whether or not to change the AI's body target direction to face the attention target.

Success/Failure

The node SUCCEEDS when the animation has finished playing, or if the animation failed to be initialized.

Example

```
<Animate name="LookAround" loop="1" />
```

AnimationTagWrapper

Adds an animation tag to the execution of a child node and clears it at the end.

Parameters

name
Animation tag to be set.

Success/Failure

The node returns the result of the execution of its child node.

Example

```
<AnimationTagWrapper name="ShootFromHip">
    <Shoot at="Target" stance="Stand" duration="5" fireMode="Burst" />
</AnimationTagWrapper>
```

AssertCondition

Checks whether or not a specified condition is satisfied.

Parameters

condition
Condition to be checked.

Success/Failure

The node SUCCEEDS if the condition is true, otherwise it FAILS.

Example

```
<AssertCondition condition="HasTarget" />
```

AssertLua

Executes a Lua script that returns true/false and translates the return value to success/failure. The result can be used to build preconditions in the MBT.

Parameters

code
Lua script to be executed.

Success/Failure

The node SUCCEEDS if the Lua script returns a value of true, otherwise it FAILS.

Example

```
<AssertLua code="return entity:IsClosestToTargetInGroup()" />
```

AssertTime

Checks whether or not a time condition is satisfied.

Parameters

since
: Name of the time stamp to check for the condition.

isMoreThan
: Condition statement used to test whether the time stamp is greater than a specified value. Cannot be used with the parameter `isLessThan`.

isLessThan
: Condition statement used to test whether the time stamp is less than a specified value. Cannot be used with the parameter `isMoreThan`.

orNeverBeenSet
: (Optional) Boolean indicating whether or not to set the node to succeed if the time stamp was never set.

Success/Failure

The node SUCCEEDS if the time condition is true, and FAILS if it is false. If the specified time stamp was not previously set, the node FAILS, unless the parameter `orNeverBeenSet` is true, in which case it SUCCEEDS.

Example

```
<AssertTime since="GroupLostSightOfTarget" isLessThan="10" orNeverBeenSet="1"
/>
```

Bubble

Displays a message in a speech bubble above the AI agent. See AI Bubbles System (p. 17).

Parameters

message
: Message string to be shown in the speech bubble.

duration
: Number of seconds to display the message. Default is 0.0.

balloon
: Boolean indicating whether or not to display the message in a balloon above the AI agent. Default is true.

log
: Boolean indicating whether or not to write the message to the general purpose log. Default is true.

Success/Failure

The node SUCCEEDS immediately after having queued the message to be displayed.

Example

```
<Bubble message="MessageToBeDisplayedAndOrLogged" duration="5.0" balloon="true"
 log="true" />
```

CheckIfTargetCanBeReached

Checks whether or not the AI agent's attention target can be reached.

Parameters

mode

Target to check for. Allowed values include:

- UseLiveTarget
- UseAttentionTarget

Success/Failure

The node SUCCEEDS if the target can be reached, otherwise it FAILS.

Example

```
<CheckIfTargetCanBeReached mode="UseLiveTarget" />
```

ClearTargets

Clears the AI agent's targets information.

Parameters

None.

Success/Failure

The node always SUCCEEDS.

Example

```
<ClearTargets />
```

Communicate

Sends a request to the communication manager to play one of the AI agent's communications. See Communication System (p. 35).

Parameters

name
The name of the communication to be played.

channel
The channel on which the communication is to be set.

waitUntilFinished
(Optional) Specifies if the execution should wait for the end of the communication before finishing.

timeout
(Optional) The threshold defining the maximum amount of seconds the node will wait.

expiry
(Optional) The amount of seconds the communication can wait for the channel to be clear.

minSilence
(Optional) The amount of seconds the channel will be silenced after the communication is played.

ignoreSound
(Optional) Sets the sound component of the communication to be ignored.

ignoreAnim
(Optional) Sets the animation component of the communication to be ignored.

Success/Failure

If the node is set to wait, the node SUCCEEDS when the communication is complete. Otherwise, it SUCCEEDS once the timeout elapses.

Example

```
<Communicate name="Advancing" channel="Tactic" expiry="1.0" waitUntilFinished="0"
 />
```

ExecuteLua

Executes a Lua script.

Parameters

code
Script to be executed.

Success/Failure

The node always SUCCEEDS.

Example

```
<ExecuteLua code="entity:SetEyeColor(entity.EyeColors.Relaxed)" />
```

GroupScope

Makes execution of a child node conditional on entering the AI agent in a group scope. Groups allow a limited number of concurrent users.

Parameters

name
Name of the group scope to enter.

allowedConcurrentUsers
(Optional) Maximum number of simultaneous users allowed in the specified group scope.

Success/Failure

The node FAILS if the AI agent cannot enter the group scope; otherwise, it returns the result of executing the child node.

Example

```
<GroupScope name="DeadBodyInvestigator" allowedConcurrentUsers="1">
    <SendTransitionSignal name="GoToPrepareToInvestigateDeadBody" />
</GroupScope>
```

IfCondition

Executes a child node if a specified condition is satisfied.

Parameters

condition
Condition statement to be checked.

Success/Failure

If the condition is satisfied, the node returns the result of executing the child node. If the condition is not satisfied, the node FAILS.

Example

```
<IfCondition condition="TargetVisible">
    <Communicate name="AttackNoise" channel="BattleChatter" expiry="2.0" wai
tUntilFinished="1" />
</IfCondition>
```

IfTime

Executes a child node if a time condition is satisfied.

Parameters

since
Name of the time stamp to check for the condition.

isMoreThan
Condition statement test whether the time stamp is greater than a specified value. Cannot be used with the parameter `isLessThan`.

isLessThan
Condition statement test whether the time stamp is less than a specified value. Cannot be used with the parameter `isMoreThan`.

orNeverBeenSet
(Optional) Boolean indicating whether or not to set the node to succeed if the time stamp was never set.

Success/Failure

If the time condition is true, the node returns the result of executing the child node. It FAILS if the time condition is false. If the specified time stamp was not previously set, the node FAILS, unless the parameter `orNeverBeenSet` is true, in which case it SUCCEEDS.

Example

```
<IfTime since="FragGrenadeThrownInGroup" isMoreThan="5.0" orNeverBeenSet="1">
    <ThrowGrenade type="frag" />
</IfTime>
```

Log

Adds a message to the AI agent's personal log.

Parameters

message
Message to be logged.

Success/Failure

The node always SUCCEEDS.

Example

```
<Log message="Investigating suspicious activity." />
```

Look

Adds a location for the AI agent to look at, and clears it when the node stops executing.

Parameters

at
Location to look at. Allowed values are:
- ClosestGroupMember
- RefPoint
- Target

Success/Failure

This node does not succeed or fail.

Example

```
<Look at="ClosestGroupMember" />
```

LuaGate

Executes a child node only if the result from running a Lua script is true.

Parameters

code
Lua script to be executed.

Success/Failure

The node SUCCEEDS if the result of the Lua script is true, and FAILS if the result is not true. On success, the node returns the result of executing the child node.

Example

```
<LuaGate code="return AI.GetGroupScopeUserCount(entity.id, 'DeadBodyInvestigat
or') == 0">
    <Animate name="AI_SearchLookAround" />
</LuaGate>
```

LuaWrapper

Runs a Lua script before and/or after the execution of a child node.

Parameters

onEnter
(Optional) Script to be executed at the start.

onExit
(Optional) Script to be executed at the end.

Success/Failure

The node returns the result of executing the child node.

Example

```
<LuaWrapper onEnter="entity:EnableSearchModule()" onExit="entity:DisableSearch
Module()">
    <Animate name="AI_SearchLookAround" />
</LuaWrapper>
```

MonitorCondition

Continuously checks the state of a specified condition.

Parameters

condition
Specifies the condition to be checked.

Success/Failure

The node SUCCEEDS when the condition is satisfied.

Example

```
<MonitorCondition condition="TargetVisible" />
```

Move

Moves the AI agent from its current position to a specified destination. If the destination is a target, then the end position is updated if it is not reached when the target moves. See Movement System (p. 32).

Parameters

speed
Speed of movement. Allowed values include:

- Walk
- Run
- Sprint

stance
Body stance while moving. Allowed values include:

- Relaxed
- Alerted
- Stand (default)

bodyOrientation
Direction the AI agents body should face during the move. Allowed values include:

- FullyTowardsMovementDirection
- FullyTowardsAimOrLook
- HalfwayTowardsAimOrLook (default)

moveToCover
Boolean indicating whether or not the AI agent is moving into cover. Default is false.

turnTowardsMovementDirectionBeforeMovingx
Boolean indicating whether or not the AI agent should first turn to the direction of movement before actually moving. Default is false.

strafe
Boolean indicating whether or not the AI agent is allowed to strafe. Default is false.

glanceInMovementDirection
Boolean indicating whether or not the AI agent can glance in the direction of movement. If false, the AI agent will always look at its look-at target. Default is false.

to
Movement destination. Allowed values include:

- Target - Current attention target.
- Cover - Current cover position.
- RefPoint - Current reference position.
- LastOp - Position of the last successful position-related operation.

stopWithinDistance
Distance from the target that the AI agent can stop moving. Default is 0.0.

stopDistanceVariation
Maximum additional distance that may be randomly added to the value of `stopDistanceVariation`, in the range [0, `stopDistanceVariation`]. Setting this value causes the stop distance to vary randomly between different executions of the node. Default is 0.0.

fireMode
Firing style while moving. Allowed values are listed for the Shoot (p. 71) node.

avoidDangers
Boolean indicating whether or not the AI agent should avoid dangers while moving. Default is true.

avoidGroupMates
Boolean indicating whether or not the AI agent should avoid group mates while moving. Default is true.

considerActorsAsPathObstacles
Boolean indicating whether or not an AI agent's pathfinder should avoid actors on the path. Default is false.

lengthToTrimFromThePathEnd
Distance that should be trimmed from a pathfinder path. Use positive values to trim from the path end , or negative values to trim from the path start. Default is 0.0.

Success/Failure

The node SUCCEEDS if the destination is reached. The node FAILS if the destination is deemed unreachable.

Example

```
<Move to="Target" stance="Alerted" fireMode="Aim" speed="Run" stopWithinDis
tance="3" />
```

Priority & Case

Prioritizes to selects from a set of possible child nodes to execute. Within a `<Priority>` node, each child node is listed inside a `<Case>` node, which defines a condition statement. A child node is selected and executed based on (1) the first child to have its condition met, and (2) in the case of ties, the order the child nodes are listed in. All but the last child must have a condition statement; the last child listed is the default case, so it's condition must always be true.

Parameters

The `<Priority>` node has no parameters.

The `<Case>` node has the following parameters:

condition
Condition statement used to prioritize a child node.

Success/Failure

The node returns the result of the executed child node.

Example

```
<Priority>
    <Case condition="TargetInCloseRange and TargetVisible">
<Melee target="AttentionTarget" />
    </Case>
    <Case>
<Look at="Target" />
    </Case>
</Priority>
```

PullDownThreatLevel

Lower's the AI agent's perception of the target's threat.

Parameters

to

Success/Failure

The node always SUCCEEDS.

Example

```
<PullDownThreatLevel to="Suspect" />
```

QueryTPS

Performs a TPS query to find a tactical position for the AI agent, and waits for a result. See AI Tactical Point System (p. 19).

Parameters

name
: Name of the TPS query to run.

register
: Location to store the result of the TPS query. Allowed values include:
  - RefPoint
  - Cover (default)

Success/Failure

The node SUCCEEDS if the TPS returns a tactical position, or FAILS if it does not find a tactical position.

Example

```
<QueryTPS name="queryName" register="Cover" />
```

RandomGate

Executes a child node (or not) based on random chance.

Parameters

opensWithChance
: Probability to use to determine whether the child node will be executed. Allowed values include floats 0.0 to 1.0.

Success/Failure

The node FAILS if the child node is not executed. If it is executed, the node SUCCEEDS AND returns the result of the execution of its child node.

Example

```
<RandomGate opensWithChance="0.5">
    <ThrowGrenade type="frag" />
</RandomGate>
```

SendTransitionSignal

Sends a signal, destined for a state machine node on the behavior tree, with the explicit intent of causing a change of state.

Parameters

name
: Name of the signal to be sent.

Success/Failure

This node does not succeed or fail.

Example

```
<SendTransitionSignal name="LeaveSearch" />
```

SetAlertness

Sets the AI agent's alertness level.

Parameters

value
Alertness level. Allowed values include integers 0 to 2.

Success/Failure

The node always SUCCEEDS.

Example

```
<SetAlertness value="1" />
```

Shoot

Sets the AI agent to shoot at a target or location.

Parameters

duration
Length of time (in seconds) the AI agent should continue shooting.

at
Location to shoot at. Allowed values include:

- AttentionTarget
- ReferencePoint
- LocalSpacePosition

fireMode
Firing style. Allowed values include:

- Off - Do not fire (default).
- Burst - Fire in bursts at living targets only.
- Continuous - Fire continuously at living targets only.
- Forced - Fire continuously at any target.
- Aim - Aim only at any target.
- Secondary - Fire secondary weapon (grenades, etc.).
- SecondarySmoke - Fire smoke grenade.
- Melee - Melee.
- Kill - Shoot at the target without missing, regardless of the AI agent's aggression/attackRange/accuracy settings.
- BurstWhileMoving - Fire in bursts while moving and too far away from the target.
- PanicSpread - Fire randomly in the general direction of the target.
- BurstDrawFire - Fire in bursts in an attempt to draw enemy fire.
- MeleeForced - Melee without distance restrictions.
- BurstSwipe - Fire in burst aiming for a head shot.

- AimSweep - Maintain aim on the target but don't fire.
- BurstOnce - Fire a single burst.

stance
Body stance while shooting. Allowed values include:
- Relaxed
- Alerted
- Crouch
- Stand

position
(Required if the target is a local space position) Local space position to be used as the target.

stanceToUseIfSlopeIsTooSteep
(Optional) Alternative stance style if the slope exceeds a specified steepness. Allowed values are the same as for `stance`.

allowedSlopeNormalDeviationFromUpInDegrees
(Optional) Maximum allowed steepness (in degrees of inclination above horizontal) to set the primary stance. At positions that exceed this slope, the alternative stance is used.

aimObstructedTimeout
(Optional) Length of time (in seconds) the AI agent's aim can be obstructed before the node will fail.

Success/Failure

The node SUCCEEDS if it executes for the specified duration. The node FAILS if the aim is obstructed for longer than the specified timeout.

Example

```
<Shoot at="Target" stance="Crouch" fireMode="Burst" duration="5" allowedSlopeNor
malDeviationFromUpInDegrees="30" stanceToUseIfSlopeIsTooSteep="Stand" />
```

ShootFromCover

Sets the AI agent to shoot at the target from cover and adjusts its stance accordingly.

Parameters

duration
Length of time (in seconds) the node should execute.

fireMode
Firing style. Allowed values are listed for the Shoot (p. 71) node.

aimObstructedTimeout
(Optional) Length of time (in seconds) the AI agent's aim can be obstructed before the node will fail.

Success/Failure

The node SUCCEEDS if it executes for the specified duration. The node FAILS if the AI agent is not in cover, if there's no shoot posture, or if the aim is obstructed for longer than the specified timeout.

Example

```
<ShootFromCover duration="10" fireMode="Burst" aimObstructedTimeout="3" />
```

Signal

Sends a signal to the AI system. See Signals (p. 86).

Parameters

name
Name of the signal to be sent.

filter
(Optional) Signal filter to use when sending the signal, which determines which AI agents will receive it.

Success/Failure

The node always SUCCEEDS.

Example

```
<Signal name="StartedJumpAttack" />
```

SmartObjectStatesWrapper

Sets the states of certain smart objects immediately before and/or after the execution of a child node.

Parameters

onEnter
(Optional) Smart object states to set at the start.

onExit
(Optional) Smart object states to set at the end.

Success/Failure

The node returns the result of executing the child node.

Example

```
<SmartObjectStatesWrapper onEnter="InSearch" onExit="-InSearch">
    <Animate name="LookAround" />
</SmartObjectStatesWrapper>
```

Stance

Sets the stance of the AI agent.

Parameters

name
Primary stance style. Allowed values include:

- Relaxed
- Alerted
- Crouch
- Stand

stanceToUseIfSlopeIsTooSteep
(Optional) Alternative stance style if the slope exceeds a specified steepness. Allowed values are the same as for `stance`.

allowedSlopeNormalDeviationFromUpInDegrees
(Optional) Maximum allowed steepness (in degrees of inclination above horizontal) to set the primary stance. At positions that exceed this slope, the alternative stance is used.

Success/Failure

The node always SUCCEEDS.

Example

```
<Stance name="Crouch" allowedSlopeNormalDeviationFromUpInDegrees="30" stanceT
oUseIfSlopeIsTooSteep="Stand" />
```

StopMovement

Sends a request to the Movement system to stop all movements. See Movement System (p. 32).

> **Note**
> This may not immediately stop the AI agent The Movement system may be dependent on animations and physics that dictate a 'natural' stop rather than an immediate cessation of movement.

Parameters

waitUntilStopped
Boolean indicating whether or not the node should wait for the Movement System to finish processing the request.

waitUntilIdleAnimation
Boolean indicating whether or not the node should wait until the Motion_Idle animation fragment begins running in Mannequin.

Success/Failure

The node SUCCEEDS if the stop request has been completed.

Example

```
<StopMovement waitUntilStopped="1" waitUntilIdleAnimation="0" />
```

Teleport

Moves the AI agent when both the destination point and source point are outside the camera view.

Parameters

None.

Success/Failure

The node always SUCCEEDS.

Example

```
<Teleport />
```

ThrowGrenade

Triggers the AI agent to attempt a grenade throw.

Parameters

timeout
: Maximum length of time (in seconds) to wait for the grenade to be thrown.

type
: Grenade type to throw. Allowed values include:
  - emp
  - frag
  - smoke

Success/Failure

The node SUCCEEDS if a grenade is thrown before it times out, otherwise the node FAILS.

Example

```
<ThrowGrenade type="emp" timeout="3" />
```

WaitUntilTime

Executes until a time condition is satisfied.

Parameters

since
: Name of the time stamp to check for the condition.

isMoreThan
: Condition statement used to test whether the time stamp is greater than a specified value. Cannot be used with the parameter `isLessThan`.

isLessThan
: Condition statement used to test whether the time stamp is less than a specified value. Cannot be used with the parameter `isMoreThan`.

succeedIfNeverBeenSet
: (Optional) Boolean indicating whether or not to set the node to succeed if the time stamp was never set.

Success/Failure

The node SUCCEEDS if the time condition is true. If the specified time stamp was not previously set, the node FAILS, unless the parameter `succeedIfNeverBeenSet` is true, in which case it SUCCEEDS.

Example

```
<WaitUntilTime since="BeingShotAt" isMoreThan="7" />
```

CryAction Nodes

These nodes provide MBT functionality for CryAction features.

AnimateFragment

Plays a Mannequin animation fragment and waits until the animation finishes.

Parameters

name
Name of the animation to play.

Success/Failure

The node SUCCEEDS if the animation is correctly played or if no operation was needed. The node FAILS if an error occurs while trying to queue the animation request.

Example

```
<AnimateFragment name="SomeFragmentName" />
```

Game Nodes

These nodes offer game-specific MBT functionality. These allow a game with multiple character types to trigger specific logic and perform actions involving each type's peculiarities. Game-specific nodes not likely to be good for "general use" will probably need customization for each game.

Character types are defined in a Lua file, which contains a table of settings for game nodes.

InflateAgentCollisionRadiusUsingPhysicsTrick

Enlarges an AI agent's capsule radius for collisions with a player. This node employs a trick in the physics system inflate the capsule radius for agent-player collisions while leaving the radius unchanged for collisions between the agent and the world.

Note
This trick is entirely isolated within this node. The node does not clean up after itself, so the capsule remains inflated after it has been used.

This trick works as follows:

1. Sets the player dimensions with the agent-vs.-player collision radius. The physics system is multi-threaded, so there's a short wait while until the player dimensions are committed.
2. Periodically inspects the player dimensions to check that the agent-vs.-player collision radius has been successfully committed. This can sometimes fail to happen, such as when the AI agent is in a tight spot and can't inflate.
3. Once the agent-vs.-player radius has been committed, goes into the geometry and sets the capsule's radius in place, using the agent-vs.-world radius. This will not affect the agent-vs.-player dimensions.

Parameters

radiusForAgentVsPlayer
Size of capsule to use when calculating collisions between the AI agent and the player.

radiusForAgentVsWorld
Size of capsule to use when calculating collisions between the AI agent and the world.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues running until it is out of the scope of the executed nodes.

Example

```
<InflateAgentCollisionRadiusUsingPhysicsTrick radiusForAgentVsPlayer="1.0" ra
diusForAgentVsWorld="0.5" />
```

KeepTargetAtADistance

Keeps the live target at a distance by physically pushing the target away when it is within a specified distance. This node is useful when there is some sort of action close to the player and you want to avoid clipping through the camera. Use of this node is preferable over increasing the AI agent's capsule size, which will also affect how the character fits through tight passages. This node is generally used in parallel with other actions that need to be performed while the player cannot come too close to the AI agent; for example, when playing an animation on the spot that can move the AI agent without moving the locator, causing camera clipping.

Parameters

distance
Minimum distance allowed between the player and the AI agent.

impulsePower
Amount of impulse used to keep the player at least at the minimum distance.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues running until it is out of the scope of the executed nodes.

Example

```
<KeepTargetAtADistance distance="1.8" impulsePower="1.5" />
```

Melee

Triggers a melee attack against the AI agent's target. The melee attack is performed if the following condition are satisfied:

- If `failIfTargetNotInNavigationMesh` is set, the target must be on a valid walkable position. Some melee animations can move the character to a position outside the navigable area if trying to melee a target outside the navigation mesh.
- If the target is not within the threshold angle specified by the entity Lua value `melee.angleThreshold`.

Parameters

target
Target of the melee attack. This parameter could be set with the AI agent's AttentionTarget or a generic RefPoint.

cylinderRadius
Radius of the cylinder used for the collision check of the hit.

hitType
Type of hit that will be reported to the game rules. Default is CGameRules::EHitType::Melee.

failIfTargetNotInNavigationMesh
Boolean indicating whether or not the node should try to melee a target that is outside the navigation mesh.

materialEffect
Name of the material effect used when the melee attack hits the target.

Success/Failure

This node succeeds regardless of whether or not a melee attack is executed and, if it is, whether or not the attack damages the target. This is because a failure in this node is not important for behavior tree logic. If it's important for the game to react to this situation, a fail option can be added.

Example

```
<Melee target="AttentionTarget" cylinderRadius="1.5" hitType="hitTypeName" ma
terialEffect="materialEffectName" />
```

Lua table settings

The Lua table `melee` contains the following settings:

```
melee =
{
    damage = 400,
    hitRange = 1.8,
    knockdownChance = 0.1,
    impulse = 600,
    angleThreshold = 180,
},
```

damage
Amount of damage a melee attack inflicts on the target.

hitRange
Height of the cylinder used to check whether or not the melee attack can hit the target.

knockdownChance
Probability that a successful melee attack knocks down the player.

impulse
Amount of impulse applied to the player in the case of a successful melee attack.

angleThreshold
Maximum angle allowed between the AI agent's direction of movement and the direction of a path between the AI agent and the target for melee attack to be attempted.

ScorcherDeploy

Manages how the Scorcher character type handles certain activity while deploying or undeploying as part of its shooting phase. This node relies on some external Lua scripts and various signals to work properly, but is useful in obfuscating some common functionality in the AI libraries.

Before and after the node runs, the following Lua functions are called: `EnterScorchTargetPhase()` and `LeaveScorchTargetPhase()`. When the node starts running, the "ScorcherScorch" animation tag is requested by Mannequin. When the node stops , if it stops normally, the "ScorcherNormal" tag is requested again. If it is terminated prematurely, it is up to the behavior tree script to define a proper exit strategy, such as requesting the "ScorcherTurtle" tag.

On requesting animation tags, the node waits for the following animation events to be received (this ensures that the transition blend animations are not interrupted):

1. "ScorcherDeployed" – when the scorcher is ready to start firing
2. "ScorcherUndeployed" – when the scorcher is again ready to walk around

The node encapsulates the following child nodes: `RunWhileDeploying` and `RunWhileDeployed`, each of which can contain exactly one child node.

RunWhileDeploying

Causes activity to happen while the Scorcher is in the process of deploying, that is, getting ready for an attack. As an example, this node might be used to control aiming before actually shooting.

The node will continue running until one of the following events occur, after which the node will be forcefully stopped:

- `ScorcherFriendlyFireWarningModule` sends one of these signals to the entity: "OnScorchAreaClear" or OnScorchAreaNotClearTimeOut"
- Mannequin animation sequence sends a "ScorcherDeployed" signal
- An internal timeout elapses

The node does not support any parameters. The node SUCCEEDS or FAILS depending on whether the child node succeeds or fails. The node is allowed to SUCCEED prematurely.

RunWhileDeployed

Controls actual aiming and firing during an attack. Duration and execution of the attack is controlled via this node.

The node does not support any parameters. The node SUCCEEDS or FAILS depending on whether the child node succeeds or fails. The node is allowed to SUCCEED prematurely. If the node SUCCEEDS, this triggers the parent node to start the undeployment sequence.

Parameters

maxDeployDuration
Length of time (in seconds) to allow the "RunWhileDeploying" child node to run. Default is 2.0.

Success/Failure

The node SUCCEEDS if the entire deploy and undeploy sequence is completed. The node FAILS if either the `RunWhileDeploying` or `RunWhileDeployed` nodes FAILED.

Example

```
<ScorcherDeploy maxDeployDuration="1.0">
    <RunWhileDeploying>
        <SomeChildNode>
    </RunWhileDeploying>
    <RunWhileDeployed>
        <SomeOtherChildNode>
    </RunWhileDeployed>
</ScorcherDeploy>
```

SuppressHitReactions

Enables or disables the Hit Reaction system for the AI agent.

Parameters

None.

Success/Failure

The node SUCCEEDS or FAILS based on success of failure of its child node.

Example

```
<SuppressHitReactions>
    <SomeChildNode />
</SuppressHitReactions>
```

Flying Nodes

These nodes provide MBT functionality related to flying vehicles.

Hover

Causes a flying AI agent to hover at its current position.

Parameters

None.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues running until forced to terminate.

Example

```
<Hover />
```

FlyShoot

Allows the AI agent to shoot at its attention target when possible from its current position.

If the AI agent's secondary weapon system is used, the node will only open fire if the weapons are able to hit close enough to the target. Otherwise normal firing rules are applied.

Parameters

useSecondaryWeapon
Boolean indicating whether or not the secondary weapon system (such as rocket launchers) should be used. Default is 0.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, the AI agent continues to shoot until forced to terminate.

Example

```
<FlyShoot useSecondaryWeapon="1" />
```

WaitAlignedWithAttentionTarget

Waits until the AI agent is facing its attention target.

Parameters

toleranceDegrees
: Maximum angle (in degrees) between the attention target and the forward direction of the AI agent to consider the AI agent to be "facing" the attention target. Allowed values include the range [0.0,180.0]. Default is 20.0.

Success/Failure

The node SUCCEEDS if the angle between the AI agent's forward direction and its attention target is within the allowed range. The node FAILS if the AI agent has no attention target.

Example

```
<WaitAlignedWithAttentionTarget toleranceDegrees="40" />
```

Fly

Allows an AI agent to fly around by following a path. Paths should be assigned to the AI agent using Flow Graph.

Parameters

desiredSpeed
: Speed of movement (in meters per second) along the path to move along the path. Default is 15.0.

pathRadius
: Radius of the path (in meters). While flying, the AI agent tries to stay within this distance from the path's line segments. Defaults is 1.0.

lookAheadDistance
: Distance (in meters) to look forward along the path for 'attractor points' to fly to. Default is 3.0.

decelerateDistance
: Distance (in meters) from the end of the path that the AI agent starts to decelerate. Default is 10.0.

maxStartDistanceAlongNonLoopingPath
: Maximum distance (in meters) to look ahead for the closest point to link with another path. This parameter is used to link with non-looping paths; for example, it is useful to prevent the AI agent from snapping to the new path at a position that seems closer but is actually behind a wall after a U-turn. Defaults is 30.0.

loopAlongPath
: Boolean indicating whether or not the AI agent should follow a path in an endless loop. Default is 0.

startPathFromClosestLocation
: Boolean indicating at what point the AI agent should start following a path. Default is 0.

  - 1 - at its closest position
  - 2 - at the first path waypoint

pathEndDistance
: Distance (in meters) from the end of the path that this node should start sending arrival notification events. Defaults is 1.0.

goToRefPoint
Boolean indicating whether or not the current reference point should be appended to the end of the path. Default is 0.

Success/Failure

The node SUCCEEDS if the AI agent reached the end of the path. The node FAILS if no valid path was assigned to the AI agent.

Example

```
<Fly lookaheadDistance="25.0" pathRadius="10.0" decelerateDistance="20.0" pa
thEndDistance="1" desiredSpeed="15" maxStartDistanceAlongNonLoopingPath="30"
loopAlongPath="0" goToRefPoint="1" startPathFromClosestLocation="1" />
```

Lua table settings

The following properties in the AI agent's Lua script table can override the default XML tags. This will allow for changes to be made at run-time through (Flow Graph) scripting.

When	Lua variable	XML tag
Each node tick	`Helicopter_Speed`	`desiredSpeed`
Node activation	`Helicopter_Loop`	`loopAlongPath`
Node activation	`Helicopter_StartFromClosestLocation`	`startPathFromClosestLocation`

Upon arrival, the following events will be emitted:

- ArrivedCloseToPathEnd
- ArrivedAtPathEnd

FlyForceAttentionTarget

Keeps an attention target on a flying vehicle by force. The attention target is acquired during each tick of the node from the `Helicopter_ForcedTargetId` Lua script variable. When the node is deactivated, a ForceAttentionTargetFinished event is emitted.

Parameters

None.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues to force the attention target until deactivation.

Example

```
<FlyForceAttentionTarget />
```

FlyAimAtCombatTarget

Aims a flying AI agent at its target, taking into account special aiming adjustments for weapons.

Parameters

None.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues to force the AI agent to rotate its body towards the attention target until termination.

Example

```
<FlyAimAtCombatTarget />
```

HeavyShootMortar

Controls shooting the mortar (or Heavy X-Pak) weapon. It tries to simplify and centralize the pre-condition check and initialization of the weapon, plus re-selection of the primary weapon.

Parameters

to
(Optional) Shooting target. Allowed values include:
- Target (default)
- Refpoint

fireMode
(Optional) Type of firing. Allowed values include:
- Charge (default)
- BurstMortar

timeout
(Optional) Maximum time (in seconds) to continue shooting. Default is 5.0.

aimingTimeBeforeShooting
(Optional) Time (in seconds) to spend aiming before starting to shoot. Value must be longer than the global timeout. Default is 1.0.

minAllowedDistanceFromTarget
(Optional) Minimum distance (in meters) to the target required to start shooting. Default is 10.0.

Success/Failure

The node FAILS if the weapon is closer to the target than the value of `minAllowedDistanceFromTarget`. The node FAILS if there are obstructions less than two meters in front of the weapon; a cylinder check is done to avoid this. The node FAILS if the timeout is reached. The node SUCCEEDS if the shooting SUCCEEDS.

Example

```
<HeavyShootMortar to="RefPoint" fireMode="Charge" aimingTimeBeforeShooting="2"
 timeout="7" />
```

SquadScope

Makes execution of a child node conditional on adding the AI agent to a squad scope. Squads allow a limited number of concurrent users.

> **Note**
> The dynamic squad system uses the AI system's cluster detector. This tool is used with `AISquadManager` to group AI agents into dynamic squads.

Parameters

name
: Name of the squad scope to enter.

allowedConcurrentUsers
: (Optional) Maximum number of simultaneous users allowed in the specified squad scope. Default is 1.

Success/Failure

The node SUCCEEDS when the child SUCCEEDS. The node FAILS if the AI agent can't enter the squad scope or if the child FAILS.

Example

```
<SquadScope name="SomeScopeName" allowedConcurrentUsers="5">
    <SomeChildNode />
</SquadScope>
```

SendSquadEvent

Sends an event to squad members only.

> **Note**
> The dynamic squad system uses the AI system's cluster detector. This tool is used with `AISquadManager` to group AI agents into dynamic squads.

Parameters

name
: Name of the event to be sent.

Success/Failure

The node always SUCCEEDS after sending the event.

Example

```
<SendSquadEvent name="SomeEventName" />
```

IfSquadCount

Makes execution of a child node conditional on whether or not the number of squad members meets a specified condition. Although all parameters are optional, at least one parameter must be used.

Note
The dynamic squad system uses the AI system's cluster detector. This tool is used with `AISquadManager` to group AI agents into dynamic squads.

Parameters

isGreaterThan
(Optional) Condition statement used to test whether the number of squad members exceeds a specified value.

isLesserThan
(Optional) Condition statement used to test whether the number of squad members is under a specified value.

equals
(Optional) Condition statement used to test whether the number of squad members exactly equals a specified value.

Success/Failure

The node SUCCEEDS if the number of squad members satisfies the specified condition statement, and FAILS if not.

Example

```
<IfSquadCount isGreaterThan="1">
    <SomeChildNode />
</IfSquadCount>
```

Refpoints

A refpoint, or reference point, is a special AI object used by goalpipes. It primarily specifies a position and, as needed, a direction. The following examples illustrate how refpoints are used.

Example 1: Updating a refpoint involving sub-goalpipes

In this example, a refpoint position is set, and a goalpipe is created containing three goalops: Locate, Stick, and Signal. Using the refpoint, Locate sets a value called LASTOP, which is used in Stick to pinpoint a destination.

Notice that the goalop Stick is defined as "+stick". This ensures that Stick is grouped with the previous goalop (Locate). As a result, if the interrupting goalpipe affects values that Stick depends on (such as LASTOP), it will return to the appropriate goalop to update the dependent values.

```
ACT_GOTO = function(self, entity, sender, data)
    if (data and data.point) then
        AI.SetRefPointPosition(entity.id, data.point);

        -- use dynamically created goal pipe to set approach distance
        g_StringTemp1 = "action_goto"..data.fValue;
        AI.CreateGoalPipe(g_StringTemp1);
        AI.PushGoal(g_StringTemp1, "locate", 0, "refpoint");
       AI.PushGoal(g_StringTemp1, "+stick", 1, data.point2.x, AILASTOPRES_USE,
 1, data.fValue); -- noncontinuous stick
       AI.PushGoal(g_StringTemp1, "signal", 0, 1, "VEHICLE_GOTO_DONE", SIGNAL
FILTER_SENDER);
        entity:InsertSubpipe(AIGOALPIPE_SAMEPRIORITY, g_StringTemp1, nil,
data.iValue);
    end
end,
```

Example 2: Using an AI anchor to set a refpoint

In this example, the Smart Object system spots a relevant AI anchor using `OnBiomassDetected`. This anchor is used to set both the position and direction of the refpoint. As a result, the AI agent walks to the refpoint, turns to the indicated direction, and then selects the next goalpipe.

```
OnBiomassDetected = function(self, entity, sender, data)
    entity:SetTargetBiomass(sender);
    entity:SelectPipe(0, "AlienTick_ReachBiomass");
end,
```

```
function AlienTick_x:SetTargetBiomass(biomass)
    self.AI.targetBiomassId = biomass.id;
    AI.SetRefPointPosition(self.id, biomass:GetWorldPos());
    AI.SetRefPointDirection(self.id, biomass:GetDirectionVector(1));
end
```

```
<GoalPipe name="AlienTick_ReachBiomass">
    <Speed id="Walk"/>
    <Locate name="refpoint"/>
    <Stick distance="0.3" useLastOp="true"/>
    <Signal name="OnBiomassReached"/>
</GoalPipe>
```

```
OnBiomassReached = function(self, entity)
    entity.actor:SetForcedLookDir(AI.GetRefPointDirection(entity.id));
    entity:SelectPipe(0, "AlienTick_CollectBiomass");
end,
```

Note
The tag <Group> was not used in this example because this particular goalpipe is not intended to be interrupted (which is not generally the case).

Signals

The Lumberyard AI system includes a fully customizable Signal system that enables AI entities to communicate with each other. Communication consists of signal events that can be sent by an AI agent to another single agent (including itself), or to a group of AI agents currently active in the game.

This topic describes how to send and receive signals between AI agents.

Signals Reference (p. 89)

Sending Signals

Signals are sent from an AI agent's behavior to one or more other AI agents using the method `AI:Signal()`.

```
AI:Signal(Signal_filter, signal_type, *MySignalName*, sender_entity_id);
```

Signal_filter
Group of AI agents to receive the signal. Allowed values include:

- 0 – AI agent specified with the entity_id parameter (usually but not always the sender itself).
- SIGNALFILTER_LASTOP – AI agent's last operation target (if it has one).
- SIGNALFILTER_TARGET – AI agent's current attention target.
- SIGNALFILTER_GROUPONLY – All AI agents in the sender's group (same group id) within communication range.
- SIGNALFILTER_SUPERGROUP – All AI agents in the sender's group (same group id) within the whole level.
- SIGNALFILTER_SPECIESONLY – All AI agents of the sender's species within communication range.
- SIGNALFILTER_SUPERSPECIES – All AI agents of the sender's species within the whole level.
- SIGNALFILTER_HALFOFGROUP – Half the AI agents in the sender's group, randomly selected.
- SIGNALFILTER_NEARESTGROUP – Nearest AI agent in the sender's group.
- SIGNALFILTER_NEARESTINCOMM – Nearest AI agent in the sender's group within communication range.
- SIGNALFILTER_ANYONEINCOMM – All AI agents within communication range.
- SIGNALID_READIBILITY – Special signal used to make the recipient perform a readability event (sound/animation).

signal_type
Type of signal, which determines how the recipient will process it. Allowed values include:

- 1 – Recipient processes signal only if it is enabled and not set to "ignorant" (see `AI:MakePuppetIgnorant`).
- 0 – The entity receiving the signal will process it if it's not set to ignorant.
- -1 – The entity receiving the signal will process it unconditionally.

MySignalName
The actual identifier of the signal. It can be any non-empty string; for the signal recipient, it must exist a function with the same name either in its current behavior, its default behavior or in the `Scripts/AI/Behaviors/Default.lua` script file in order to react to the received signal.

entity_id
The entity id of the signal's recipient. Usually you may want to put entity.id (or self.id if it's called from the entity and not from its behavior), to send the signal to the sender itself, but you can also put any other id there to send the signal to another entity.

Receiving Signals

The action to be performed once a signal is received is defined in a function like this:

```
MySignalName = function(self, entity, sender)
```

self
The recipient entity's behavior.

entity
The recipient entity.

sender
The signal's sender.

This function is actually a callback which, exactly like the system events, can be defined in the recipient entity's current behavior, the default idle behavior (if it's not present in current behavior) or in the `Scripts/AI/Behaviors/Default.lua` script file (if not present in the default idle behavior).

As for system events, a signal can be used also to make a character change its behavior; if we add a line like the following in a character file:

```
Behaviour1 = {
    OnEnemySeen   = *Behaviour1*,
    OnEnemyMemory = *Behaviour2*,
    &#8230;
    MySignalName  = *MyNewBehaviour*,
}
```

This means that if the character is currently in Behaviour1, and receives the signal MySignalName, after having executed the callback function above it will then switch its behavior to MyNewBehaviour.

Signal Example

A typical example is when a player's enemy spots the player: its OnEnemySeen system event is called, and let's suppose he wants to inform his mates (The guys with his same group id). In his default idle behavior (i.e., `CoverAttack.lua` if the character is Cover), we modify its OnEnemySeen event like this:

```
OnEnemySeen = function( self, entity, fDistance )
    -- called when the enemy sees a living enemy

    AI:Signal(SIGNALFILTER_GROUPONLY, 1, "ENEMY_SPOTTED",entity.id);
end,
```

Here we have defined a new signal called ENEMY_SPOTTED.

The next step is to define the callback function. Let's assume the other members in the group have the same character, we then add the callback function to the same idle behavior in which we have just modified OnEnemySeen.

```
ENEMY_SPOTTED = function (self, entity, sender)
    entity:Readability("FIRST_HOSTILE_CONTACT");
    entity:InsertSubpipe(0, "DRAW_GUN");
End,
```

This will make the guys (including the signal sender itself, who has the same behavior) change their animation and producing some kind of alert sound (readability), and then draw their gun. Notice that by modifying its idle behavior, we create a default callback which will be executed for any behavior the character is in. Later on, we may want to override this callback in other behaviors. For example, if we wanted the character to react differently whether it's in idle or attack behavior, we'll add the following callback function in the `CoverAttack.lua` file:

```
ENEMY_SPOTTED = function (self, entity, sender)
    entity:SelectPipe(0, "cover_pindown");
End,
```

Where "cover_pindown" is a goalpipe that makes the guy hide behind the nearest cover place to the target.

We can extend this to other characters: if there are group members with different characters (i.e. Scout, Rear etc) and we want them to react as well, we must add the ENEMY_SPOTTED callback also to their idle/attack behavior. Finally, we want the guys to switch their behavior from idle to attack if they see an enemy.

We'll then add the following line to the character (`Scripts/AI/Characters/Personalities/Cover.lua` in the example):

```
CoverIdle = {
    &#8230;
    ENEMY_SPOTTED = *CoverAttack*,
},
```

Behavior Inheritance

If specific signals are to be used in more than one behavior, there is an inheritance mechanism. Behavior classes can either directly inherit a more general implementation by keyword `Base = [CRYENGINE:ParentBehaviorName]` or indirectly, as a character's Idle behavior as well as the default behavior (defined in file DEFAULT.lua) are considered as fallback behaviors if a signal is not implemented in the current behavior.

Signals Reference

A typical signal handler looks something like this:

```
OnEnemySeen = function(self, entity, distance)
    -- called when the AI sees a living enemy
end,
```

Parameters self (behavior table) and entity (entity table) are passed to every signal. Additional parameters are specific to the signal being used.

See also: `\Game\Scripts\AI\Behaviors\Template.lua`.

Perception Signals

The following signals are sent to AI agents when perception types of their attention targets change.

Note that AITHREAT_SUSPECT < AITHREAT_INTERESTING < AITHREAT_THREATENING < AITHREAT_AGGRESSIVE.

No Target

Name	Parameters	Description
OnNoTarget		Attention target is lost

Sound

Sound heard (no visible target).

Name	Parameters	Description
OnSuspectedSound-Heard		Threat is AITHREAT_SUSPECT

Name	Parameters	Description
OnInterestingSound-Heard		Threat is AITHREAT_INTERESTING
OnThreateningSound-Heard		Threat is AITHREAT_THREATENING
OnEnemyHeard		Threat is AITHREAT_AGGRESSIVE

Memory

The target is not visible and is in memory.

Name	Parameters	Description
OnEnemyMemory		Threat is AITHREAT_THREATENING
OnLostSightOfTarget		Threat is AITHREAT_AGGRESSIVE
OnMemoryMoved		Threat is AITHREAT_AGGRESSIVE and its location or owner has changed

Visual

The target is visible.

Name	Parameters	Description
OnSuspectedSeen		Threat is AITHREAT_SUSPECT
OnSomethingSeen		Threat is AITHREAT_INTERESTING
OnThreateningSeen		Threat is AITHREAT_THREATENING
OnEnemySeen	distance	Threat is AITHREAT_AGGRESSIVE
OnObjectSeen	distance, data	AI sees an object registered for this signal. data.iValue = AI object type (e.g. AIOBJECT_GRENADE or AIOBJECT_RPG)
OnExposedToExplosion	data	AI is affected by explosion at data.point
OnExplosionDanger		Destroyable object explodes

Awareness of Player

Name	Parameters	Description
OnPlayerLooking	sender, data	Player is looking at the AI for entity.Properties.awarenessOfPlayer seconds. data.fValue = player distance
OnPlayerSticking	sender	Player is staying close to the AI since <entity.Properties.awarenessOfPlayer> seconds

Name	Parameters	Description
OnPlayerLookingAway	sender	Player has just stopped looking at the AI
OnPlayerGoingAway	sender	Player has just stopped staying close to the AI

Awareness of Attention Target

Name	Parameters	Description
OnTargetApproaching		
OnTargetFleeing		
OnNewAttentionTarget		
OnAttentionTar-getThreatChanged		
OnNoTargetVisible		
OnNoTargetAware-ness		
OnSeenByEnemy	sender	AI is seen by the enemy

Weapon Damage

Name	Parameters	Description
OnBulletRain	sender	Enemy is shooting
OnDamage	sender, data	AI was damaged by another friendly/unknown AI. data.id = damaging AI's entity id
OnEnemyDamage	sender, data	AI was damaged by an enemy AI. data.id = dam-aging enemy's entity id

Proximity

Name	Parameters	Description
OnCloseContact		enemy gets at a close distance to an AI (defined by Lua Property "damageRadius" of this AI)
OnCloseCollision		

Vehicles

Name	Parameters	Description
OnVehicleDanger	sender, data	vehicle is going towards the AI. data.point = vehicle movement direction, data.point2 = AI direction with respect to vehicle

Name	Parameters	Description
OnEndVehicleDanger		
OnTargetTooClose	sender, data	attention target is too close for the current weapon range (it works only if AI is a vehicle)
OnTargetTooFar	sender, data	attention target is too close for the current weapon range (it works only if AI is a vehicle)
OnTargetDead		

User-defined

Custom signals can be sent when an attention target enters or leaves certain ranges. This is configured using the following Lua functions:

```
AI.ResetRanges(entityID);
AI.AddRange(entityID,range, enterSignal, leaveSignal);
AI.GetRangeState(entityID, rangeID);
AI.ChangeRange(entityID, rangeID, distance);
```

Weapon-Related Signals

Name	Parameters	Description
OnLowAmmo		
OnMeleeExecuted		
OnOutOfAmmo		
OnReload		AI goes into automatic reload after its clip is empty
OnReloadDone		reload is done
OnReloaded		

Navigation Signals

Pathfinding

Name	Parameters	Description
OnEndPathOffset	sender	AI has requested a path and the end of path is far from the desired destination
OnNoPathFound	sender	AI has requested a path which is not possible
OnPathFindAtStart		
OnBackOffFailed	sender	AI tried to execute a "backoff" goal which failed
OnPathFound	sender	AI has requested a path and it's been computed successfully

Steering

Name	Parameters	Description
OnSteerFailed		

Smart Objects

Name	Parameters	Description
OnEnterNavSO		
OnLeaveNavSO		
OnUseSmartObject		

Navigation Shapes

Name	Parameters	Description
OnShapeEnabled		
OnShapeDisabled		

Tactics Signals

Tactical Point System

Name	Parameters	Description
OnTPSDestNotFound		
OnTPSDestFound		
OnTPSDestReached		

Cover

Name	Parameters	Description
OnHighCover		
OnLowCover		
OnMovingToCover		
OnMovingInCover		
OnEnterCover		
OnLeaveCover		
OnCoverCompromised		

Groups Signals

Name	Parameters	Description
OnGroupChanged		
OnGroupMemberMutilated		
OnGroupMemberDiedNearest		

Formation

Name	Parameters	Description
OnNoFormationPoint	sender	AI couldn't find a formation point
OnFormationPointReached		
OnGetToFormationPointFailed		

Group Coordination

Group target is the most threatening target of the group.

Name	Parameters	Description
OnGroupTargetNone		
OnGroupTargetSound		
OnGroupTargetMemory		
OnGroupTargetVisual		
PerformingRole		

Flow Graph Signals

These are signals sent by corresponding Flow Graph nodes when they are activated.

Name	Parameters	Description
ACT_AIMAT		AI:AIShootAt
ACT_ALERTED		AI:AIAlertMe
ACT_ANIM		AI:AIAnim
ACT_ANIMEX		AI:AIAnimEx
ACT_CHASETARGET		Vehicle:ChaseTarget
ACT_DIALOG		AI:ReadabilityDialog (also sent by Dialog System)
ACT_DIALOG_OVER		Sent by Dialog System
ACT_DUMMY		
ACT_DROP_OBJECT		AI:AIDropObject
ACT_ENTERVEHICLE		Vehicle:Enter
ACT_EXECUTE		AI:AIExecute
ACT_EXITVEHICLE		Vehicle:Exit, Vehicle:Unload
ACT_FOLLOW		AI:AIFollow
ACT_FOLLOWPATH		AI:AIFollowPath, AI:AIFollowPathSpeedStance, Vehicle:FollowPath
ACT_GRAB_OBJECT		AI:AIGrabObject
ACT_GOTO		AI:AIGoto, AI:AIGotoSpeedStance, and the AI Debugger when the user clicks the middle mouse button.
ACT_JOININFORMATION		AI:AIFormationJoin
ACT_SHOOTAT		AI:AIShootAt
ACT_USEOBJECT		AI:AIUseObject
ACT_VEHICLESTICKPATH		Vehicle:StickPath
ACT_WEAPONDRAW		AI:AIWeaponDraw
ACT_WEAPONHOLSTER		AI:AIWeaponHolster
ACT_WEAPONSELECT		AI:AIWeaponSelect

Other Signals

Forced Execute

Name	Parameters	Description
OnForcedExecute		
OnForcedExecuteComplete		

Animation

Name	Parameters	Description
AnimationCanceled		

Game

Name	Parameters	Description
OnFallAndPlay		

Vehicle-related

Name	Parameters	Description
OnActorSitDown	Actor has entered a vehicle	

Squads

Name	Parameters	Description
OnSomebodyDied		
OnBodyFallSound		
OnBodyFallSound		
OnUnitDied		
OnSquadmateDied		
OnPlayerTeamKill		
OnUnitBusy		
OnPlayerDied		

Name	Parameters	Description
OnFriendInWay	sender	AI is trying to fire and another friendly AI is on his line of fire
URPRISE_ACTION		
OnActionDone	data	AI action of this agent was finished. data.ObjectName is the action name, data.iValue is 0 if action was cancelled or 1 if it was finished normally, data.id is the entity id of "the object" of the AI action

Animation

This section describes Lumberyard's Animation system. It includes discussions of key concepts and provides information on working with the system programmatically.

This section includes the following topics:

Animation Overview

One of Lumberyard’s goals is to push the boundaries of animations, which are all rendered in real time. Lumberyard provides tools to create both linear and interactive animations:

- Linear animation is the kind of animation seen in movies and cut-scenes, which play as a video.
- Interactive animation is used to convey AI and avatar (player) behavior, with sequences dependent on player choices in game play.

There is a big difference between how each type of animation is incorporated into a game, although this difference may not be obvious to the player, who simply sees characters moving on-screen. The key difference is in the decision-making process: who decides what a character on the screen is going to do next?

Linear Animations

In linear animation, the decision-making process happens inside the head of the people designing the animation. During this process, an animator has direct control over every single keyframe. They don’t need to deal with collision detection, physics and pathfinding; characters only run into walls or collide with each other when the animator wants them to. AI behavior does not need to react to player behavior; the person who writes the storyboard decides how intelligent or stupid the characters are. To show interactions between characters, you can put them in motion-capture suits and record their performances.

A linear animation sequence needs to show action from a single camera angle because the audience won't be moving during the animation; as a result, animators don't need to deal with transitions and motion combinations; they control every aspect of the motion clip. Because everything is fixed and predicable, it's possible to guarantee a consistent motion quality. Animators can always go back and adjust details in the scene, such as add or delete keyframes, adjust the lighting, or change the camera position.

The technical challenges with creating linear animation primarily involve rendering issues, such as not dropping the frame rate and ensuring that facial and body animations are in sync.

All linear animations in Lumberyard are created with Track View Editor.

Interactive Animations

Creating interactive animations presents significantly tougher challenges. Animators and programmers do not have direct control over a character's on-screen movements. It is not always obvious where and how the decision-making process happens. It is usually a complex combination of AI systems, player input, and sometimes contextual behavior.

By definition, interactive animation is responsive. It looks visibly different depending on an individual user's input and adapts automatically to actions on the screen. Moving from linear animation to interactive animation requires more than just a set of small tweaks or a change in complexity—it requires a completely different technology under the hood. With interactive animation, an animator cannot precisely plan and model a character's behavior. Instead, animators and programmers develop a system that allows them to synthesize motion automatically and define rules for character behavior.

Automatic motion synthesis is a crucial feature in making animation more interactive. A system that synthesizes motion must be very flexible, because it is difficult to predict the sequence of actions that a character may take, and each action can start at any time.

Imagine, for example, a character moving through an outdoor environment. At a minimum, the designer needs to specify the style, speed, and direction of the character's locomotion. There should also be variations in motion while running uphill or downhill, leaning when running around corners or carrying objects of different sizes and weights—the character should run faster while carrying a pistol than when hefting a rocket launcher. It might also be necessary to interactively control emotional features such as happiness, anger, fear, and tiredness. Additionally, the character may need to perform multiple tasks simultaneously, such as walking in one direction, turning head and eyes to track a bird in another direction, and aiming a gun at a moving object in third direction. Providing unique animation assets for every possible combination and degree of freedom is nearly impossible and would involve an incredibly large amount of data. A mechanism for motion modifications is needed to keep the asset count as low as possible.

Developing such a system involves close collaboration and a tight feedback loop between programmers, animators, and designers. Problems with the behavior and locomotion systems (either responsiveness or motion quality) are usually addressed from several sides.

Interactive animation can be divided into two categories: **Avatar control** and **AI control**. In both cases, animators and programmers have indirect control over the actual behavior of a character in game play, because decision making for the character's next action happens elsewhere. Let's take a closer look at the situation in game environments.

Avatar control

An avatar character is controlled by the game player, whose decisions determine all of the avatar's actions. The locomotion system takes the player's input and translates it on the fly into skeleton movements (using procedural and data-driven methods). With avatar control, high responsiveness is the top priority, while motion quality might be limited by the game rules. This means that many well-established rules for 'nice'-looking animations are in direct conflict with the responsiveness you need for certain types of game play.

The quality of animations as executed on the screen depends largely on the skills and decisions of each player controlling the character—they decide what the avatar will do next. Because a player's actions are unpredictable, motion planning based on predictions is not possible. Complex emotional control is not possible (and probably not needed). It's only possible on a raw level, such as soft punch versus an aggressive punch. However, it might be possible to let the player control the locomotion of the avatar, and to let the game code control the emotional behavior of the avatar by blending in "additive animations" based on the in-game situation.

In all these scenes, the player is controlling the character with a game pad. The character's presentation on the screen is using animation assets created by animators.

AI control

For AI characters, the decision-making process happens entirely inside the game code. Game developers design a system to generate behavior, which acts as an intermediary between the game creators and players. For the system to perform this task, it is necessary for game designers to explicitly specify behavioral decisions and parameters for AI characters, including a clear definition of the rules of movements for each character type. Interactive animation for AI characters is much harder to accomplish than animations for avatars, but at the same time it offers some (not always obvious) opportunities to improve motion quality. High responsiveness is still the primary goal but, because character choices happen inside the game code, it is possible in certain circumstances to predict a character's actions. If the AI system knows what the AI character wants to do next, then it is possible to incorporate this knowledge into motion planning. With good motion planning, interactive animation might be able to use more classical or 'nice' animation rules. As a result, AI control can have a somewhat higher motion quality than avatar control, though at the cost of having more complex technology under the hood.

The only source of uncertainty in such a prediction system is the player: the AI reacts to the player, and predicting the player's actions is impossible. As a result, it's nearly impossible to create the right assets for every in-game situation, and this in turn makes it impossible to guarantee a consistent motion quality. For an animator working on interactive animation, it can be a significant problem to have no direct control over the final animation—it's never clear when the work is complete. This is one reason why the linear animation in movies and cut-scenes look superior, and why interactive animations can be troublesome.

Lumberyard tackles the problem with interactive animation in multiple levels:

- In the low-level CryAnimation system library, the engine provides support for animation clips, parametrized animation, and procedural modification of poses. Animations can be sequenced together or layered on top of each other in a layered transition queue.
- In the high-level CryAction library, the CryMannequin system helps to manage the complexity of animation variations, transitions between animations, animations that are built up out of many others, sequencing of procedural code, links to game code, and so on.

Scripted Animations

Because interactive animation is much more difficult than linear animation, many games blur the line between cut-scenes and in-game actions by using interactive scripted sequences.

In this case, characters act on a predefined path. The quality of this kind of motion can be very high. Because it is not fully interactive, animators have more control over the entire sequence, a kind of manually designed motion planning. These are perfectly reasonable cheats to overcome hard-to-solve animation problems. It may be even possible to script the entire AI sequence to allow near-cut-scene quality. The action feels interactive and looks absolutely cinematic, but it is actually more an illusion of interactivity.

In the game Crysis, Crytek designers made use of scripted animations in many scenes. In the "Sphere" cut-scene, the Hunter is shown walking uphill and downhill and stepping over obstacles. This is a scripted sequence where the assets were made for walking on flat ground, but Crytek used CCD-IK to adapt the character's legs to the uneven terrain. In the "Fleet" cut-scene with the Hunter on the carrier deck, the player can move around while the Hunter is fighting other non-playing characters.

Both scenes look and feel highly interactive but they are not. The Hunter doesn't respond to the player and the player cannot fight the Hunter. The scenes are fully linear and scripted, basically just animated background graphics. These sequences were created in Track View Editor. Some of them used the Flow Graph Editor. When the cut-scene is over, the Hunter turns into an AI-controlled interactive character.

Animation Events

Animations in Lumberyard can be marked up to send custom events at a specific time in an animation. This markup is used for time-aligned blending; for example, to match footplants in animations. Another application of animation events is to spawn particle effects at the right moment.

These events can also be used by a variety of systems that need to receive information about when an animation has reached a certain point, such as in combination with a melee system.

Marking Up Animations with Events

Events for animations are stored in an XML file that is loaded when the character starts up. For this to happen automatically, the database must be included in the `chrparams` file.

Receiving Animation Events in the Game Code

Animation events are passed on to the game object once they have been triggered. The Actor and Player implementations both handle these animation events. See either `Actor.cpp` or `Player.cpp` for the function:

```
void AnimationEvent(ICharacterInstance *pCharacter, const AnimEventInstance
&event)
```

Limb IK Technical

Lumberyard's animation system allows the setup of IK chains for characters.

When an IK chain is active, the system calculates the joint angles in the chain so that the end effector (typically a hand or foot) reaches the target position.

Setting Up

IK chains are defined in the `chrparams` file.

Using LimbIK from Code

To activate a Limb IK chain from outside the Animation system, use the function `SetHumanLimbIK`, accessible through the `ISkeletonPose` interface. The `SetHumanLimbIK` function needs to be called in each frame in which you want the IK chain to be active. The name of the Limb IK chain is defined in the `chrparams` file:

```
ISkeletonPose& skeletonPose = ...;
skeletonPose.SetHumanLimbIK(targetPositionWorldSpace, "RgtArm01");
```

Animation Streaming

Animation is very memory-intensive and tends to use a large amount of resources. Limited memory budgets, high numbers of animated joints, and requirements for high animation quality make it wasteful for a project to keep all animations constantly loaded in memory.

Lumberyard's animation system alleviates this issue by streaming in animation resources (file granularity level) when needed, and unloading them when not needed. Streaming of asset files is achieved by using the DGLINK Streaming System. Streaming assets in and out allows the system to keep only the needed resources in memory—which is done at the expense of complexity, as you must now plan how and when animation resources are used.

Animation Data

Animation data usage is divided into two main sections:

- The **header** section contains generic information for an animation (filename, duration, flags, etc).
- The **controller** section contains the animation curves. For each joint involved, this section contains information on all the position and orientation values that the joint needs in order to play that animation. Even when compressed, controller data can easily take up more than 95% of the total memory required for an animation.

Animation Header Data

Header data for animations is stored in `CAF` files and in the `animations.img` file.

`CAF` files contain the header information on a single animation, while `animations.img` contains header information for all animations in the build. The `animations.img` is obtained as a result of processing all the animations with the Resource Compiler.

The engine usually loads all the animation files' headers from the `animations.img` file instead of loading from individual files (reading the information from individual files can considerably slow down loading time).

Because of the extreme size difference between controllers and headers, Lumberyard streams only the controller data in and out of memory. The header data for all animations is kept at all times in memory, as it is practical to have that information available at all times.

Note
During development—for example, when working with local animation files—you must disable usage of `animations.img` and load the header information from individual `CAF` files instead. To do so, set the `ca_UseIMG_CAF` console variable to 0 before the engine starts.

Animation Controller Data

The controller data for animations is stored in CAF files or DBA files.

- `CAF` files contain controller information for a single animation.
- `DBA` files contain controller information for a group of animations.

When a DBA is loaded, controllers for all animations contained in that DBA are available until the DBA is unloaded. For this reason, it is useful to group animations that are used together in the same DBA. An extra benefit of putting similar animations together in a DBA is that equal controllers are only stored once. This reduces the memory usage of your animations.

Loading Controller Data

The animation system properly plays animations only when their controllers are in memory.

If controller data is not available when playback of an asset is requested, the animation system streams it in from disk. Streaming of controller data is performed asynchronously—the animation system does not wait until after asset playback is requested. This prevents stalling the system.

If high level systems fail to notify the animation system that they require controller data (see the preload functions section), the animation system does not know that an asset is required until it is requested to play. This is dangerously close to when the controller data is needed. If the controller data is not available in time, it typically leads to visual glitches, which can sometimes be observed, for example, only the first time an animation is played.

Therefore, it is important to have controller data streamed in before playback of an animation is requested. This minimizes undesired glitches that occur while waiting for animation streaming to end.

The amount of time required for streaming to complete depends on many factors, such as the current system load, streaming speed of the target system, size of the resource that needs to be loaded, and so on.

Unloading Controller Data

The animation system will not unload controller data that is currently in use.

It is possible to prevent unloading of animation data entirely by setting `ca_DisableAnimationUnloading` to 1.

Controllers in `CAF` files are unloaded after the system detects that they are no longer in use. To prevent controllers in CAF files from being unloaded, set `ca_UnloadAnimationCAF` to 0.

Controllers in DBA files remain in memory until a certain amount of time passes after the animations in them are used. However, if the DBA is locked, controllers are not unloaded until the lock status is set back to 0.

To change the time that the animation system waits to unload controllers in DBA files, use the following `cvars`:

- `ca_DBAUnloadUnregisterTime` – Timeout in seconds after the last usage of a controller and all animations using that DBA; when this timeout is reached, the DBA marks their controller data as 'unloaded'.
- `ca_DBAUnloadRemoveTime` – Timeout in seconds after the last usage of a controller in a DBA; when this timeout is reached, the DBA performs an actual unload from memory. This value should be greater than or equal to `ca_DBAUnloadUnregisterTime`.

The following section describes how to lock individual resources in memory to prevent the system from unloading them.

Preloading and Keeping Controllers in Memory

Preload functions are performed by high level systems or user code (usually game code), as these contain most of the information on when and how assets are accessed. For example, trackview looks a number of seconds ahead in the timeline for any animations that appear, and calls the preload functions.

Preloading Controllers in DBA files

To preload and trigger the streaming of a `DBA` file:

```
gEnv->pCharacterManager->DBA_PreLoad(dbaFilename, priority);
```

To trigger the streaming of a `DBA` file, and request a change to the locked state (which specifies whether it should be locked in memory):

```
gEnv->pCharacterManager->DBA_LockStatus(dbaFilename, lockStatus, priority);
```

To unload all controller data in a DBA from memory (unloads data only if none of the controllers are currently being used):

```
gEnv->pCharacterManager->DBA_Unload(dbaFilename);
```

Note
To make the system automatically load and lock a DBA file while a character is loaded, use the `flags="persistent"` in the `chrparams` file.

Preloading Controllers in CAF files

To increase the reference count of a `CAF` file:

```
gEnv->pCharacterManager->CAF_AddRef(lowercaseAnimationPathCRC);
```

Controllers for a `CAF` file start streaming in when its reference count goes from 0 to 1.

To decrease the reference count of a `CAF` file:

```
gEnv->pCharacterManager->CAF_Release(lowercaseAnimationPathCRC);
```

Controllers for a `CAF` file are unloaded by the animation system only after the reference count reaches 0 (the animation system, when playing a CAF file, also increases this reference count, so that an animation is not unloaded while in use).

To check whether the controllers for a `CAF` file are loaded:

```
gEnv->pCharacterManager->CAF_IsLoaded(lowercaseAnimationPathCRC);
```

To synchronously load the controllers for a CAF file:

```
gEnv->pCharacterManager->CAF_LoadSynchronously(lowercaseAnimationPathCRC);
```

Synchronously loading CAF assets is strongly discouraged unless absolutely necessary, as it will likely result in stalls.

Animation Debugging

Several tools are available for debugging animation issues.

Layered Transition Queue Debugging

You can enable on-screen debug information to see which animations are queued and playing, as well as information about the applied pose modifiers and IK.

Show Per Entity

To show the transition queue for all the character instances of a specified entity:

```
es_debuganim <entityname> [0 | 1]
```

<entityname>
Name of the entity to debug. In a single player game, the player is typically called "dude." Note that the GameSDK example player has both a first person and a third person character instance.

[0 | 1]
Specify 1 or no second parameter to turn it on for this specific entity. Specify 0 to turn it off.

Examples

To turn on debugging for a player with the entity name "dude":

```
es_debuganim dude 1
```

To turn off debugging for an entity called "npc_flanker_01":

```
es_debuganim npc_flanker_01 0
```

Show Per CharacterInstance

You can show the transition queue for all character instances or the ones that have a specific model name.

```
ca_debugtext [<modelname-substring> | 1 | 0]
```

<modelname-substring>
Shows information for all character instances whose modelname contains the specified string.

[0 | 1]
If 1 is specified, all character instances are shown. If 0 is specified, the debug text is turned off.

Examples

To show information on all character instances with "player" in their model name:

```
ca_debugtext player
```

To turn off all transition queue information:

```
ca_debugtext 0
```

Interpreting the Output

Each animation in the transition queue is displayed as in the following example. Key elements of this display are described following the example.

```
AnimInAFIFO 02: t:1043 _stand_tac_idle_scar_3p_01 ATime:0.84 (1.17s/1.40s) AS
pd:1.00 Flag:00000042 (----------I-K----) TTime:0.20 TWght:1.00 seg:00 inmem:1
(Try)UseAimIK: 1 AimIKBlend: 1.00 AimIKInfluence: 1.00 (Try)UseLookIK: 0 Look
IKBlend: 0.00 LookIKInfluence: 0.00
MoveSpeed: 4.49 locked: 1
PM class: AnimationPoseModifier_OperatorQueue, name: Unknown
...
LayerBlendWeight: 1.00
...
ADIK Bip01 RHand2RiflePos_IKTarget: 0.24 Bip01 RHand2Aim_IKTarget: 1.00 Bip01
LHand2Aim_IKTarget: 0.00
```

Text Color

- When an animation is not yet active, it is in black or green.
- When an animation is active, it is in red or yellow.

Or in detail:

- Red Channel = Animation Weight
- Green Channel = (layerIndex > 0)
- Alpha Channel = (Weight + 1)*0.5

AnimInAFIFO Line (one per animation)

```
AnimInAFIFO 02: t:1043 _stand_tac_idle_scar_3p_01 ATime:0.84 (1.17s/1.40s) AS
pd:1.00 Flag:00000042 (----------I-K----) TTime:0.20 TWght:1.00 seg:00 inmem:1
```

AnimInAFIFO 02
: Layer index (decimal, zero-based)

t:1043
: User token (decimal)

_stand_tac_idle_scar_3p_01
: Animation name (alias) of the currently playing animation, aim/look-pose or bspace

ATime:0.84 (1.17s/1.40s)
: ATime:XXXX (YYYYs/ZZZZs)
  - XXXX = Current time in 'normalized time' (0.0...1.0) within the current segment
  - YYYY = Current time (seconds) within the current segment
  - ZZZZ = Expected duration (seconds) of the current segment

ASpd:1.00
: Current animation speed (1.0 = normal speed)

Flag:00000042 (----------I-K----)

Animation Flags

Flag:XXXXXXXX (+ybVFx3nSIAKTRLM)

The first number is the animation flags in hexadecimal

Between parentheses you see the individual flags:

char	flag	value			
+	CA_FORCE_TRANS-ITION_TO_AN-IM	0x008000			
y	CA_FULL_ROOT_PRI-ORITY	0x004000			
b	CA_RE-MOVE_FROM_FIFO	0x002000			
V	CA_TRACK_VIEW_EX-CLUSIVE	0x001000			
F	CA_FORCE_SKEL-ETON_UP-DATE	0x000800			
x	CA_DIS-ABLE_MUL-TILAYER	0x000400			
3	CA_KEY-FRAME_SAMPLE_30Hz	0x000200			
n	CA_AL-LOW_AN-IM_RESTART	0x000100			
S	CA_MOVE2IDLE	0x000080			
I	CA_IDLE2MOVE	0x000040			
A	CA_START_AFTER	0x000020			
K	CA_START_AT_KEY-TIME	0x000010			
T	CA_TRANS-ITION_TIME-WARPING	0x000008			
R	CA_RE-PEAT_LAST_KEY	0x000004			
L	CA_LOOP_AN-IMATION	0x000002			
M	CA_MANU-AL_UPDATE	0x000001			

TTime:0.20
Transition Time

Total length of transition into this animation in seconds (this is static after pushing the animation)

TWght:1.00
Transition Weight

Current weight of this animation within the transition (0 = not faded in yet, 1 = fully faded in)

seg:00
Current segment index (zero-based)

inmem:1
Whether or not the animation is in memory (0 basically means it's not streamed in yet)

Aim/Look-IK Line

```
(Try)UseAimIK: 1 AimIKBlend: 1.00 AimIKInfluence: 1.00 (Try)UseLookIK: 0 Look
IKBlend: 0.00 LookIKInfluence: 0.00
```

(Try)UseAimIK: 1
Whether Aim IK is turned on or not (set using PoseBlenderAim::SetState)

AimIKBlend: 1.00
Weight value requested for Aim IK (could go up and down based on fade times, etc.)

AimIKInfluence: 1.00
Final influence weight value of AimIK (== smoothed(clamped(AimIKBlend)) * weightOfAllAimPoses)

(Try)UseLookIK: 0
Whether Look IK is turned on or not

LookIKBlend: 0.00
Weight value requested for Look IK (could go up and down based on fade times, etc.)

LookIKInfluence: 0.00
Final influence weight value of LookIK (== smoothed(clamped(LookIKBlend)) * weightOfAllLookPoses)

Parameter Line(s) (only for blend spaces)

```
MoveSpeed: 4.500000 locked: 1
TravelAngle: 0.000000 locked: 0
```

MoveSpeed: 4.500000
Value for the specified blend space parameter (MoveSpeed in this case)

locked: 1
Whether or not the parameter is locked (= unable to change after it is set for the first time)

PoseModifier Lines (if running)

```
PM class: AnimationPoseModifier_OperatorQueue, name: Unknown
```

Displays which pose modifiers are running in this layer. Shows the class as well as the name (if available).

LayerBlendWeight Line (not on layer 0)

```
LayerBlendWeight: 1.00
```

The weight of this layer (0.00 - 1.00)

ADIK Line(s) (only if animation driven IK is applied)

```
ADIK Bip01 RHand2RiflePos_IKTarget: 0.24 Bip01 RHand2Aim_IKTarget: 1.00 Bip01
LHand2Aim_IKTarget: 0.00
```

Displays a list of the animation driven IK targets and their current weight. For more detailed position/rotation information, use the separate cvar `ca_debugadiktargets 1`.

CommandBuffer Debugging

At the lowest level, the animation system executes a list of simple commands to construct the final skeleton's pose.

These commands are, for example, "sample animation x at time t, and add the result with weight w to the pose". Or "clear the pose".

To enable on-screen debug information to see what is pushed on the command buffer (for all characters), use the following command:

```
ca_debugcommandbuffer [0 | 1]
```

Warning Level

To control when the animation system produces warnings using the ca_animWarningLevel cvar:

```
ca_animWarningLevel [0 | 1 | 2 | 3]
```

0
: Non-fatal warnings are off.

1
: Warn about illegal requests.

  For example, requesting to start animations with an invalid index.

2
: Also warn about things like 'performance issues.'

  For example, animation-queue filling up. This might 'spam' your console with a dump of the animation queue at the time of the issue.

3 (default)
: All warnings are on. This includes the least important warnings; for example, a warning when playing uncompressed animation data.

Fall and Play

"Fall and Play" activates when a character is ragdollized (on an interface level, it is called `RelinquishCharacterPhysics`) with a >0 stiffness. This activates angular springs in the physical ragdoll that attempts to bring the joints to the angles specified in the current animation frame. The character also tries to select an animation internally based on the current fall and play stage. If there are none, or very few, physical contacts, this will be a falling animation; otherwise it will be the first frame of a standup animation that corresponds to the current body orientation.

Standup is initiated from outside the animation system through the fall and play function. During the standup, the character physics is switched back into an alive mode and his final physical pose is blended into a corresponding standup animation. This, again, is selected from a standup anims list to best match this pose.

Filename convention for standup animations: When an animation name starts with "standup", it is registered as a standup animation. Also, a type system exists which categorizes standup animations by the string between "standup_" and some keywords ("back", "stomach", "side"). You can control which type to use with `CSkeletonPose::SetFnPAnimGroup()` methods. At runtime, the engine checks the most similar standup animation registered to the current lying pose and blends to it.

Some example filenames:

- `standUp_toCombat_nw_back_01`
- `standUp_toCombat_nw_stomach_01`

While the character is still a ragdoll, it is also possible to turn off the stiffness with a `GoLimp` method.

Time in the Animation System

The Animation system uses different units of 'time,' depending on the system. How those units of time compare is best explained using an example.

The definition of 'frames': The Animation system uses a fixed rate of 30 frames per second (fps). Of course, games can run at higher frame rates, but some operations in the Editor that use the concept of 'frames'—or operations that clamp the animation duration to 'one frame'—assume a frame rate of 30 fps.

Assume then that you have an animation with a duration of 1.5 seconds. This means that the animation has 46 frames (note that this includes the final frame). So, in the case of Real Time, assume an animation starts at time 0, has no segmentation, and is played back at normal speed. However, rather than using Real Time, the Animation system typically uses Animation Normalized Time. This is compared with Real Time in the following table:

Frame Index	Real Time (seconds)*	Animation Normalized Time**
0	0.0 s	0.0
1	0.033.. s = 1/30 s	0.022.. = 1/45
..	..	..
30	1.0 s	0.666.. = 30/45
..	..	..
44	1.466.. s = 44/30 s	0.977.. = 44/45

Frame Index	Real Time (seconds)*	Animation Normalized Time**
45	1.5 s = 45/30 s	1.0

* Real time is used to define duration:

- Duration = lastFrame.realTime - firstFrame.realTime. That's 1.5s in our example.
- `IAnimationSet::GetDuration_sec()` returns the duration of an animation.

  Note: For a parametric animation, this returns only a crude approximation—the average duration of all its examples, ignoring parameters or speed scaling.
- `CAnimation::GetExpectedTotalDurationSeconds()` returns the duration of an animation that is currently playing back.

  Note: For a parametric animation, this returns only a crude approximation, assuming the parameters are the ones that are currently set and never change throughout the animation.
- No function exists that returns the Real Time of an animation. To calculate that, you must manually multiply Animation Normalized Time with the duration.

** Animation Normalized Time:

- Time relative to the total length of the animation.
- Starts at 0 at the beginning of the animation and ends at 1 (= RealTime/Duration = Keytime/LastKeyTime).
- Used by functions such as `ISkeletonAnim::GetAnimationNormalizedTime()` and `ISkeletonAnim::SetAnimationNormalizedTime()`.
- Is not well-defined for parametric animations with examples that have differing numbers of segments. For more information, see the following section, Segmentation.

Segmentation

In practice, the animation system does not use Animation Normalized Time; this terminology was used to make the introduction easier to understand. Typically, Segment Normalized Time is used. To understand Segment Normalized Time, you must first understand segmentation.

For time warping (phase matching) purposes, animations can be split into multiple segments. For example, to time warp from a walk animation with 2 cycles to a walk animation with 1 cycle, you have to annotate the first animation and split it into two (these are segments). To achieve this segmentation, you must add a segment1 animation event at the border between the cycles.

Note
An animation without segmentation has exactly 1 segment, which runs from beginning to end.

Segmentation introduces a new unit for time, Segment Normalized Time, which is time relative to the current segment duration.

Extending our example further, observe what happens when a segment1 animation event at 1.0s is added to split the animation into two segments.

Frame Index	Real Time	AnimEvents	(Animation) Normalized Time	Segment Index*	Segment Normalized Time**
0	0.0 s		0.0	0	0.0

Frame Index	Real Time	AnimEvents	(Animation) Normalized Time	Segment Index*	Segment Normalized Time**
1	0.033.. s		0.022..	0	0.033.. = 1/30
..	..		..	..	..
30	1.0 s	segment1	0.666..	1	0.0
..	..		..	..	..
44	1.466.. s		0.977..	1	0.933.. = 14/15
45	1.5 s		1.0	1	1.0

* Segment index:

- Identifies which segment you are currently in. Runs from 0 to the total number of segments minus 1.
- While an animation is playing, you can use `CAnimation::GetCurrentSegmentIndex()` to retrieve it.
- When using ca_debugtext or es_debuganim, then this index is displayed after "seg:".

** Segment normalized time:

- Time relative to the current segment's duration.
- 0 at the beginning of the segment, 1 at the end (only 1 for the last segment, as you can see in the table).
- While an animation is playing, you can use `CAnimation::Get/SetCurrentSegmentNormalizedTime()` to get or set the Segment Normalized Time.
- As the names suggest, `CAnimation::GetCurrentSegmentIndex()` retrieves the current segment index and `CAnimation::GetCurrentSegmentExpectedDurationSecondsx()` retrieves the duration of the current segment.
- When representing time within parametric animations, it is more convenient to use Segment Normalized Time than Animation Normalized Time; therefore, Segment Normalized Time is used at runtime.
- AnimEvent time is specified using Animation Normalized Time (except for the special case of parametric animation; see the following section).
- When using `ca_debugtext` or `es_debuganim`, Segment Normalized Time is displayed after "ATime:". Following that, the real time within the segment and the segment duration are displayed within the parentheses.

Playback Speed

Playback speed does not impact the functions that compute duration of playing animations, such as `CAnimation::GetExpectedTotalDurationSeconds()` or `ISkeletonAnim::CalculateCompleteBlendSpaceDuration()`.

Segmented Parametric Animation

Animation Normalized Time, Segment Index, and Duration all create ambiguity for segmented parametric animations. This is because each example animation within the parametric animation can have its own number of segments. To avoid ambiguity, animation events in or on segmented parametric animations use Segment Normalized Time. As a result, an animation event will be fired multiple times (once per segment) during the animation.

- `ISkeletonAnim::GetAnimationNormalizedTime()` uses a heuristic: It currently looks for the example animation with the largest number of segments and returns the animation normalized time within that example.
- `ISkeletonAnim::GetCurrentSegmentIndex()` uses a different heuristic: It currently returns the segment index in the example animation, which happens to be the first in the list.

Given this, we are considering redefining the above based on the following observation: You can define the total number of segments in a parametric animation as the number of segments until repetition starts.

So, say you have a parametric animation consisting of 2 examples—one with 2 segments and the other with 3 segments. This will start to repeat after 6 segments (the lowest common multiple of 2 and 3). However, you can uniquely identify each possible combination of segments using any number from 0 to 5.

The Character Tool uses this method to achieve a well-defined duration. The `ISkeletonAnim::CalculateCompleteBlendSpaceDuration()` function calculates the duration until the parametric animation starts to repeat (assuming the parameters remain fixed). It reverts to the regular `GetExpectedTotalDurationSeconds()` implementation for non-parametric animations so that the function can be used in more general situations.

Animation with Only One Key

Normally your animations have at least two keys. However, when you convert these into additive animations, the first frame is interpreted as the base from which to calculate the additive, leaving only 1 frame in the additive animation (this means that, in respect to the asset, both the start and end time of the asset are set to 1/30 s).

Functions retrieving the total duration of this animation will return 0.0 (for example, `IAnimationSet::GetDuration_sec()`, `ISkeletonAnim::CalculateCompleteBlendSpaceDuration()`, and `CAnimation::GetExpectedTotalDurationSeconds()`).

However, for playback purposes, the animation system handles these animations as if they have a duration of 1/30th of a second. For example, Animation Normalized Time still progresses from 0 to 1, while real time goes from 0 to 1/30th of a second. `CAnimation::GetCurrentSegmentExpectedDurationSecondsx()` also returns 1/30th of a second in this case.

Direction of Time

Time typically cannot run backward when playing an animation. You can move time backward only if you do it manually by setting the flag `CA_MANUAL_UPDATE` on the animation and using `CAnimation::SetCurrentSegmentNormalizedTime`. See the example DGLINK `CProceduralClipManualUpdateList::UpdateLayerTimes()`.

Time within Controllers

Different units are used for controllers that contain the actual key data and are used for animation sampling.

Frame Index	Real Time	I_CAF Ticks*	Keytime**
0	0.0 s	0	0.0

Frame Index	Real Time	I_CAF Ticks*	Keytime**
1	0.033.. s	160	1.0
..	..	..	..
30	1.0 s	4800	30.0
..	..	..	..
44	1.466.. s	7040	44.0
45	1.5 s	7200	45.0

* I_CAF Ticks:

- Used within I_CAF files to represent time
- There are 4800 I_CAF ticks per second (this is currently expressed by the fact that TICKS_CONVERT = 160 in Controller.h, which assumes 30 keys/second)

** Keytime

- Used at runtime to pass time to the controllers for sampling animation
- Used within CAF files to represent time
- A floating point version of 'frame index'
- Can represent time in between frames
- Use `GlobalAnimationHeaderCAF::NTime2KTime()` to convert from Animation Normalized Time to Keytime
- All animation controllers in the runtime use Keytime

Animation assets can also have a StartTime other than 0.0s—this complicates matters slightly, but only for the controllers. Typically, for everywhere but the controllers, time is taken relative to this StartTime.

Asset Importer (Preview) Technical Overview

The Lumberyard Asset Importer and the FBX Importer are in preview release and are subject to change.

This topic provides a high-level technical overview of the Lumberyard Asset Importer architecture and its core libraries. The preview release of the Asset Importer features:

- A new FBX Importer (built entirely from scratch) that makes it easier for you to bring single static FBX meshes and single materials into Lumberyard. For information on how to use the FBX Importer tool, available from Lumberyard Editor, see FBX Importer in the Amazon Lumberyard User Guide.
- An abstraction layer that you can extend to accept other input formats.

You can use the abstraction layer to add support for older formats like OBJ and DAE, or for custom data formats. A generic layer of data called the *scene graph* enables this extensibility, as explained in the following section.

Architecture and Concepts

Following are the key concepts and components of the Lumberyard Asset Importer.

FBX – File extension for Autodesk's filmbox file format. This is a general use container file for many types of data commonly used in video game development. Most modeling packages have the ability to export the FBX file format natively. The preview release of the Lumberyard Asset Importer supports this format.

Importer – Code that takes input data creates a *scene graph* (Lumberyard's generic tree-based representation of data) or *scene manifest* from it. For example, you might implement an `FbxSceneGraphImporter` that can construct a scene graph from an FBX file. Alternatively, you could have a `JsonManifestImporter` that constructs a scene manifest from a JSON file.

Scene – This is an abstraction that contains a scene graph tree data structure and a scene manifest. The scene manifest contains metadata about the scene graph.

Scene graph – A generic tree-based representation of data. The scene graph is a layer of data between an intermediate 3D asset (like FBX) and the Lumberyard engine. A scene graph is composed internally of *scene nodes* which optionally contain data in the form of a *data object*.

Scene node – A node within a scene graph that may contain data in the form of a data object, or exist solely for hierarchical reasons. A node's name is made unique in the scene graph by concatenating all its parent nodes and delimiting them with a '.' For example, if a node named `root` has the two children `child1` and `child2`, the full node names are `root`, `root.child1`, and `root.child2`. Even if another node has a child named `child2`, the full name is unique because of the hierarchy.

Data object – A generic container for data which supports RTTI (run-time type information) and casting through the `AZ_RTTI` system. This is the basic unit of storage used by both the `SceneManifest` key-value-pair dictionary and the `SceneNode` internal data payload. The `DataObject` class provides limited *reflection* support for editing of class data, but not serialization to file.

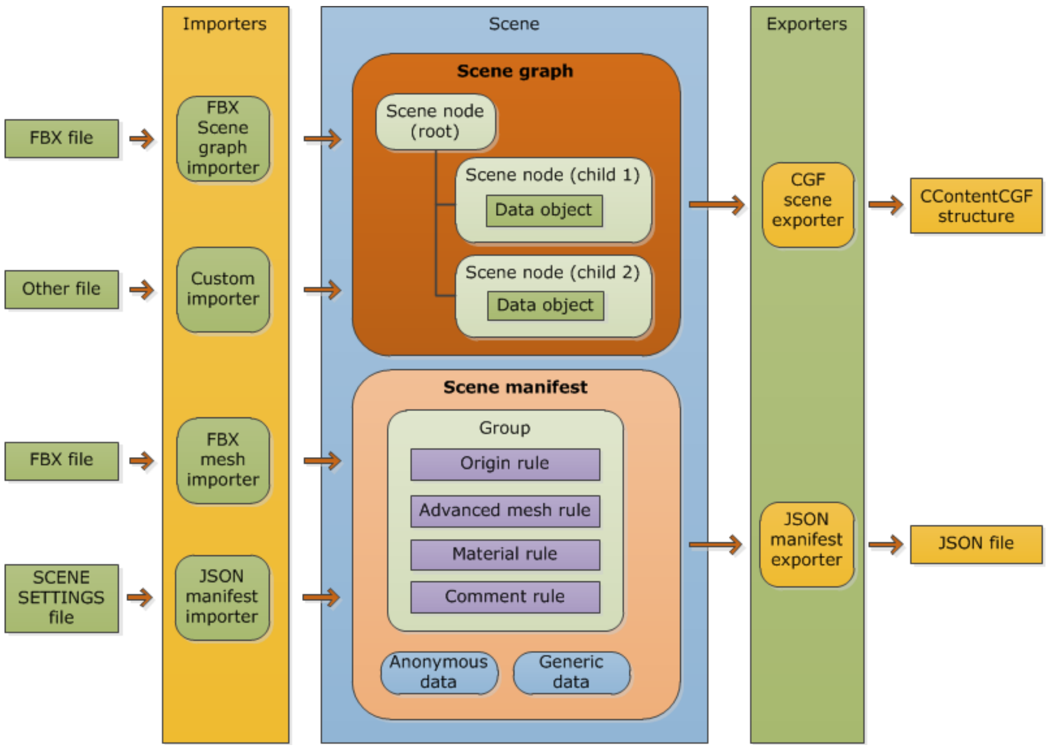
Scene manifest – This is a flat dictionary of metadata about a scene graph. This metadata will be mostly in the form of *groups* and *rules*, but supports generic and anonymous data provided by the user as well. All data is stored as key-value-pairs, with the key being a string, and the value being a `DataObject`.

Groups – These structures, managed by a scene manifest, contain rules that specify how to process scene graph data so that game data files can be generated. The current version of the Asset Importer supports *mesh groups,* which can generate CGF and MTL files.

Rules – These structures, contained in groups, specialize the handling of asset processing or display information like metadata or comments to the user. For example, you might create rules to change the vertex indexing bit count or provide positional or rotation offsets in the files that are output. The current version of the Asset Importer supports the rule types *origin*, *advanced mesh*, *material*, and *comment*.

Exporter – Code that takes data from a scene graph or scene manifest and changes it into another data format. For example, you might have a `CgfSceneExporter` that exports scene graph data using instructions from the scene manifest into the `CContentCGF` structure in memory. You could use `CgfExporter`, which allows the creation of static geometry files. You could also have a `JsonManifestExporter` that writes the data contained in a scene manifest into a JSON file on disk.

The following diagram illustrates the relationships among the components described.



Other Components

Asset Processor – The Asset Importer uses the Lumberyard Asset Processor. The asset processor runs in the background looking for changes in source data like FBX files or their metadata like `.scenesettings` files. When source or metadata files are added, removed, or changed, the asset processor automatically reprocesses the source data and outputs game data in a cache directory.

Reflection – The Lumberyard engine supports a robust and performant reflection system built on the `AZ_RTTI` system of `AzCore`. The Asset Importer and associated systems display data through the Lumberyard `ReflectedPropertyGrid` UI system that is part of the `AzToolsFramework`. The `AzToolsFramework` is a common UI framework for displaying and editing class data.

Core Libraries

Following are the core libraries of the Asset Importer. The name of each library is followed by its folder location in parentheses.

SceneCore (`\dev\Code\Tools\SceneAPI\SceneCore`) – This library contains common data storage structures, specifically: `Scene`, `SceneGraph`, `SceneManifest`, and `DataObject`. It also contains interfaces for currently supported `Group`, `Rule`, and `SceneNode` data types. It contains basic iterators that can be used to easily iterate through `SceneGraph` or `SceneManifest` data. Finally, it contains basic importers and exporters common to all use cases.

FbxSceneBuilder (`\dev\Code\Tools\SceneAPI\FbxSceneBuilder`) – This library enables the loading of FBX data and its conversion into `SceneGraph` data. For convenience, it includes the full source of the lightweight pass-through `FBXSdkWrapper`. The `FBXSdkWrapper` includes the importer for FBX data so that `SceneCore` does not require a dependency on the FBX SDK. This is the suggested model if you want to extend the `SceneAPI` codebase to support file formats that have dependencies on specialized libraries.

SceneData (`\dev\Code\Tools\SceneAPI\SceneData`) – This library contains the concrete implementations provided with the Lumberyard Asset Importer for `Group`, `Rule`, and `SceneNode` data types. These implementations are provided in a separate library that illustrates how you can extend or implement your own rule sets by using the `SceneCore` library. The `SceneData` library also contains factories for generating different types of data.

EditorAssetImporter (`\dev\Code\Sandbox\Plugins\EditorAssetImporter`) – This is the Lumberyard Editor plugin that allows the editing of scene manifests. It uses the `FbxSceneBuilder` and `SceneCore` libraries to read and write data defined in the `SceneData` library to disk when the user requests it. Currently, `EditorAssetImporter` supports only files that have the `.fbx` file extension. Other `SceneBuilder` implementations could support additional file types.

ResourceCompilerScene (`\dev\Code\Tools\rc\ResourceCompilerScene`) – The Resource Compiler (`rc.exe`) plugin uses the instructions from `SceneManifest` to process `SceneGraph` and generate game data. The Resource Compiler uses the `FbxSceneBuilder` and `SceneCore` libraries to read, write, and store data in a format defined by the `SceneData` library. The Resource Compiler is currently associated with the `.fbx` file extension. Other importers, if implemented, could translate data from non-FBX sources into `SceneGraph` data.

FBXSdkWrapper (`\dev\Code\Tools\SceneAPI\FBXSdkWrapper`) – As mentioned, this is a wrapper around the FBX SDK provided for convenience and testing. If you have an internal FBX serialization and storage model, you can use this library to replace it with the Autodesk FBX SDK.

FBX Importer Example Workflow

The following workflow illustrates how these libraries might work together in a production environment.

1. You open the Lumberyard Editor and launch the **FBX Importer**. The importer loads the `EditorAssetImporter.dll` Lumberyard Editor plugin. Note: For information on how to use the FBX Importer, see FBX Importer.
2. Using the FBX Importer file browser, you choose an FBX file that has been output from the modeling package. The FBX Importer (a specialized importer contained in `FbxSceneBuilder`) creates a `SceneGraph` that contains the equivalent of the data in the `.fbx` file.
3. If this is the first time you are importing the `.fbx` file, an empty `SceneManifest` is also created. If you have already imported the file, the `JsonManifestImporter` loads the corresponding `.scenesettings` file from the same directory and creates a new `SceneManifest`.
4. Using the Asset Importer UI, you create groups that define the output files that the Lumberyard Asset Importer will create. The groups specify names for the output files and the target nodes in the `SceneGraph` that will generate the data for the output files. You also create rules to change how the output files are processed.
5. When you have created all the desired groups and rules for the scene graph, you click **Import**. This creates or overwrites the scene manifest by saving it to a `.scenesettings` file in the same location as the `.fbx` file you loaded. For example, if `character.fbx` is loaded, a `character.scenesettings` file is created in the same directory. `JsonManifestExporter` performs this operation. If an MTL file does not exist, `MaterialExporter` generates a default MTL file.
6. The Asset Processor detects the creation of, or any change to, the `.scenesettings` file and calls the Resource Compiler (`rc.exe`) so that the `.fbx` file will be reprocessed.
7. The Resource Compiler detects the `.fbx` file type, recognizes that it has a `ResourceCompilerScene` plugin module that processes `.fbx` files, and runs the plugin.
8. The `ResourceCompilerScene` plugin creates a scene graph and scene manifest that contain data equivalent to the data in the `.fbx` file. The plugin parses the groups and rules in `SceneManifest` and

calls specialized exporters that use these groups and rules to write the scene graph data to game data files.

9. When the Resource Compiler completes, the MTL and the CGF files for the models are in the correct location and ready for use in your game.

Possible Asset Importer Customizations

Because `SceneGraph` and `SceneManifest` are flexible and extensible constructs, you can use them to easily process additional data types. The `DataObject` structure, which allows RTTI-based handling of arbitrary data, is key to this flexibility. Following are some customization examples:

- **Input File Types**: To process file types other than FBX, you implement a new importer that generates a `SceneGraph` from an arbitrary data format.
- **Input File Data**: Because the `SceneNode` in `SceneGraph` accepts arbitrary data payloads in the form of RTTI enabled `DataObject` structures, you build a `SceneGraph` structure to handle specialized or anonymous data unique to your team.
- **Custom Asset Processing Instructions**: Because `Group` and `Rule` are implemented as pure virtual interfaces, you easily customize them with instructions specific to your project.
  - Because the `SceneManifest` supports anonymous data like strings and numbers, the code you require for prototyping is kept to a minimum.
  - You extend or replace the basic concrete implementation of the `SceneData` library to fulfill your team’s specific asset processing requirements.

Callback References

This section provides script callback information for the Entity system and game rules.

This section includes the following topics:

Entity System Script Callbacks

This topic describes all callbacks for the Entity system. Use of these callbacks functions is not obligatory, but some cases require that entities behave properly within the Lumberyard Editor. For example, the `OnReset` callback should be used to clean the state when a user enters or leaves the game mode within the Lumberyard Editor.

Default State Functions

Callback Function	Description
OnSpawn	Called after an entity is created by the Entity system.
OnDestroy	Called when an entity is destroyed (like `OnShutDown()` gets called).
OnInit	Called when an entity gets initialized via ENTITY_EVENT_INIT, and when its ScriptProxy gets initialized.
OnShutDown	Called when an entity is destroyed (like `OnDestroy()` gets called).
OnReset	Usually called when an editor wants to reset the state.
OnPropertyChange	Called by Lumberyard Editor when the user changes one of the properties.

Script State Functions

Callback Function	Description
OnBeginState	Called during `Entity.GotoState()` after the state has been changed (that is, after `OnEndState()` is called on the old state).
OnBind	Called when a child entity is attached to an entity. Parameters include: • script table for the child entity
OnCollision	Called when a collision between an entity and something else occurs. Parameters include: • script table with information about the collision
OnEndState	Called during `Entity.GotoState()` while the old state is still active and before `OnBeginState()` is called on the new state.
OnEnterArea	Called when an entity has fully entered an area or trigger. Parameters include: • **areaId** (int) • **fade fraction** (float) This value is 1.0f if the entity has fully entered the area, or 0.0f in the case of trigger boxes.
OnEnterNearArea	Called when an entity enters the range of an area. Works with Box-, Sphere- and Shape-Areas if a sound volume entity is connected. Takes OuterRadius of sound entity into account to determine when an entity is near the area.
OnLeaveArea	Called when an entity has fully left an area or trigger. Parameters include: • **areaId** (int) • **fade fraction** (float) This value is always 0.0f.
OnLeaveNearArea	Called when an entity leaves the range of an area. Works with Box-, Sphere- and Shape-Areas if a sound volume entity is connected. Takes OuterRadius of sound entity into account to determine when an entity is near the area.
OnMove	Called whenever an entity moves through the world.
OnMoveNearArea	Called when an entity moves. Works with Box-, Sphere- and Shape-Areas if a sound volume entity is connected. Takes OuterRadius of sound entity into account to determine when an entity is near the area.
OnProceedFadeArea	Called when an entity has recently entered an area and fading is still in progress. Parameters include: • **areaId** (int) • **fade fraction** (float)

Callback Function	Description
OnSoundDone	Called when a sound stops. Parameters include: • **soundId** (int) The ID of the sound played, which was provided with the request to play the sound.
OnStartGame	Called when a game is started.
OnStartLevel	Called when a new level is started.
OnTimer	Called when a timer expires. Parameters include: • **timerId** (int) The ID of the time, provided by `Entity.SetTimer()` . • **period** (int) Length of time, in milliseconds, that the timer runs
OnUnBind	Called when a child entity is about to be detached from an entity. Parameters include: • script table for the child entity
OnUpdate	Called periodically by the engine on the entity's current state. This assumes the console variable `es_UpdateScript` is set to 1.

Game Rules Script Callbacks

This topic provides reference information on callbacks used with the GameRules scripts.

Callback Function	Description
OnAddTaggedEntity	Called when a player is added as a tagged player on the minimap. Called on the server only. • **shooterId** – Entity that tagged the target player. • **targetId** – Tagged player.
OnClientConnect	Called when a player connects. Called on the server only. • **channelId**
OnClientDisconnect	Called when a player disconnects. Called on the server only. • **channelId**

Callback Function	Description
OnClientEnteredGame	Called when a player enters the game and is part of the game world. Called on the server only. • **channelId** – Channel identifier of the player. • **playerScriptTable** – The player's script table. • **bReset** – Boolean indicating whether or not the channel is from the reset list. • **bLoadingSaveGame** – Boolean indicating whether or not the call was made during a saved game loading.
OnDisconnect	Called when the player disconnects on the client. Called on the client only. • **cause** – Integer identifying the disconnection cause. See `EDisconnectionCause`. • **description** – Human readable description of the disconnection cause.
OnChangeSpectatorMode	Called when a player changes the spectator mode. Called on the server only. • **entityId** – Player who made the change. • **mode** – New spectator mode (1=fixed, 2=free, 3= follow). • **targetId** – Possible target entity to spectate. • **resetAll** – Boolean indicating whether or not to reset player-related things like the inventory.
OnChangeTeam	Called when a player switches teams. Called on the server only. • **entityId** – Player who switched teams. • **teamId** – New team identifier.

Callback Function	Description
OnExplosion	Called when an explosion is simulated. Called on the server and client. • **pos** – Position of the explosion in the game world. • **dir** – Direction of the explosion. • **shooterId** • **weaponId** • **shooter** • **weapon** • **materialId** • **damage** • **min_radius** • **radius** • **pressure** • **hole_size** • **effect** • **effectScale** • **effectClass** • **typeId** • **type** • **angle** • **impact** • **impact_velocity** • **impact_normal** • **impact_targetId** • **shakeMinR** • **shakeMaxR** • **shakeScale** • **shakeRnd** • **impact** • **impact_velocity** • **impact_normal** • **impact_targetId** • **AffectedEntities** – Affected entities table. • **AffectedEntitiesObstruction** – Affected entities obstruction table.

Cloud Canvas

Cloud Canvas is a visual scripting interface to AWS services. With Cloud Canvas you can build connected game features that use Amazon DynamoDB, AWS Lambda, Amazon Simple Storage Service (Amazon S3), Amazon Cognito, Amazon Simple Notification Service (Amazon SNS), and Amazon Simple Queue Service (Amazon SQS). With Cloud Canvas, you can create cloud-hosted features such as daily gifts, in-game messages, leaderboards, notifications, server-side combat resolution, and asynchronous multiplayer gameplay (e.g. card games, word games, ghost racers, etc.). Cloud Canvas eliminates the need for you to acquire, configure, or operate host servers yourself, and reduces or eliminates the need to write server code for your connected gameplay features.

Features

Cloud Canvas offers a wide range of helpful components:

- Tools to enable a team to build a game with cloud-connected features.
- Flow graph nodes to communicate directly from within the client to AWS services such as Amazon S3, Amazon DynamoDB, Amazon Cognito, AWS Lambda, Amazon SQS, and Amazon SNS.
- Tools to manage AWS resources and permissions that determine how developers and players access them.
- Management of AWS deployments so that development, test, and live resources are maintained separately.
- Methods for players to be authenticated (anonymous and authenticated). Players can be authenticated from a variety of devices and access their game data by logging in with an Amazon, Facebook, or Google account.

Example Uses

Consider the many ways you can use Amazon Web Services for connected games:

- Store and query game data such as player state, high scores, or world dynamic content: Amazon S3 and DynamoDB
- Trigger events in real time and queue data for background processing: Amazon SQS and Amazon SNS
- Execute custom game logic in the cloud without having to set up or manage servers: AWS Lambda

- Employ a daily gift system that tracks user visits and rewards frequent visits: Amazon Cognito, Amazon S3, DynamoDB, AWS Lambda
- Present a message of the day or news ticker that provides updates on in-game events: Amazon Cognito, Amazon S3, AWS Lambda

To see how Cloud Canvas uses AWS services in a sample project, see *Don't Die* Sample Project (p. 197). For a tutorial on Cloud Canvas, see Lumberyard Tutorials.

Tools

You can access Cloud Canvas functionality by using any of the following:

- **Flow Nodes** – For designers to leverage the AWS cloud.
- **Cloud Canvas C++ APIs** – For software development.
- **Cloud Canvas Command Line (p. 170)** – For managing features, mappings, deployments, and projects.
- **Lumberyard Editor Tools (p. 169)** – For navigating directly to the AWS consoles supported by Cloud Canvas, and for managing Cloud Canvas permissions and deployments.

To see how AWS services used for the *Don't Die* sample project, see *Don't Die* Sample Project (p. 197).

Knowledge Prerequisites

You need the following to take advantage of Cloud Canvas:

- **An understanding of AWS CloudFormation Templates** – Cloud Canvas uses the AWS CloudFormation service to create and manage AWS resources. Our goal is for Cloud Canvas to minimize what you need to know about AWS CloudFormation and AWS in general.
- **Familiarity with JSON** – Cloud Canvas leverages JSON for storing configuration data, including AWS CloudFormation Templates. Currently, you'll need to be familiar with this text format to work with the Cloud Canvas resource management system. A JSON tutorial can be found here.

Topics

Pricing

Cloud Canvas uses AWS CloudFormation templates to deploy AWS resources to your account. Although there is no additional charge for Cloud Canvas or AWS CloudFormation, charges may accrue for using the associated AWS services. You pay for the AWS resources created by Cloud Canvas and AWS CloudFormation as if you created them manually. You only pay for what you use as you use it. There are no minimum fees and no required upfront commitments, and most services include a free tier.

For pricing information on the AWS services that Cloud Canvas supports, visit the following links.

Amazon Cognito Pricing

Amazon DynamoDB Pricing

AWS Lambda Pricing

Amazon S3 Pricing

Amazon SNS Pricing

Amazon SQS Pricing

To see pricing for all AWS services, visit the Cloud Services Pricing page.

To see the AWS services used for the *Don't Die* sample project, see *Don't Die* Sample Project (p. 197).

Cloud Canvas Core Concepts

Cloud Canvas helps you manage cloud resources and connect your game with the AWS cloud. Understanding its concepts will benefit anyone on your team who interacts with the cloud-connected components of your game, including designers, programmers, and testers.

This section covers the following:

- What Cloud Canvas is and how it relates to your AWS account
- The Amazon Web Services that Cloud Canvas supports
- How Cloud Canvas helps you manage your resources
- How your game can communicate with the cloud through the flow graph visual scripting system
- How a team can collaborate on a game that has cloud-connected features

Prerequisites

Before reading this topic, you should have a basic understanding of the Lumberyard engine and the flow graph system.

AWS, Cloud Canvas, and Lumberyard

Amazon Web Services (AWS) is an extensive and powerful collection of cloud-based services. You can use these services to upload or download files, access databases, execute code in the cloud, and perform many other operations. A cloud service saves you the trouble of maintaining the infrastructure that it relies on.

Cloud-Based Resources

When you want to use one of a cloud service, you do so through a resource, a cloud-based entity that is available for your use, help, or support. Resources include a database, a location for storing files, the code that a service runs, and more.

When you create a resource, it exists in the cloud, but you can use it and manage its content. You also specify the permissions that individuals or groups have to access or use the resource. For example, you might allow anyone in the public to read from your database but not write to it or modify it.

AWS Accounts

Your resources are owned by an AWS account. The AWS account allows you and your team to share access to the same resources. For example, your team's AWS account might own a database resource so that you and your teammate can both work with the same database.

You, or someone on your team, is an administrator. The administrator creates the AWS account for your team and gives individuals on the team access to the account's resources.

Lumberyard, Cloud Canvas, and Flow Graph

Cloud Canvas is a Lumberyard Gem (extension) that enables your Lumberyard games to communicate with AWS resources. To integrate the communication with Amazon Web Services directly into your game logic, you use Lumberyard's flow graph visual scripting system.

The flow graph nodes that you create use Cloud Canvas to make the actual calls from your game to AWS resources. For example, when a player's game ends, you can add flow graph nodes to submit the player's score to a high score table in the cloud. Later, you can use flow graph to call the high score table to request the top 10 scores.

Amazon Web Services Supported by Cloud Canvas

Several AWS offerings are available through Cloud Canvas that can enhance your game.

File Storage in the Cloud

For storing files in the cloud, Cloud Canvas supports Amazon Simple Storage Service (Amazon S3). Amazon S3 offers a storage resource called a bucket, which you can think of as a large folder. You can build a folder structure in an Amazon S3 bucket just like a folder on a local computer. Amazon S3 buckets have a number of uses in games, including the following:

- Storing files that your game can download. These files can be levels, characters, or other extensions for your game. You can add new files after your game has shipped. Because your game uses Cloud Canvas to download and integrate this content, your customers do not need to download a new client.
- Your game can upload user-generated content. For example, your game might take a screenshot whenever a player beats the last boss. Cloud Canvas uploads the screenshot to your bucket, and your game makes the screenshot available on a website or to other players of the game.

Databases

For storing data like a person's name, phone number, and address in the cloud, Cloud Canvas supports the Amazon DynamoDB database service. Amazon DynamoDB operates on resources called tables. These tables grow and adapt as you build and iterate your game.

Here are some ways in which you can use Amazon DynamoDB table resources in your game:

- Track account details and statistics for a player. Give each player a unique ID so that you can look up a player's hit points, inventory, gold, and friends.
- Add or remove fields to accommodate new features in your game.
- Perform data analyses. For example, you can run complex queries to find out how many players have unlocked a particular achievement.
- Manage game-wide features or events such as a querying a high score table or retrieving a message of the day.

Executing Cloud-Based Logic

For executing code in the cloud, Cloud Canvas supports the AWS Lambda service. AWS Lambda executes resources called functions. You provide the code for a Lambda function, and your game calls the Lambda service through Cloud Canvas to run the function. The Lambda service returns the data from the function to the game.

Your Lambda functions can even call other Amazon Web Services like Amazon DynamoDB and perform operations on their resources. Following are some examples:

- **Submit a high score** – A Lambda function can accept a player's ID and new score, look up the player ID in the database, compare the score with existing scores, and update the highest score if necessary.
- **Sanitize your data** – A Lambda function can check for malicious or unusual input. For example, if a player tries to upload a new high score of 999,999,999 when the best players can't reach 1,000, your Lambda function can intercept the submission and either reject it or flag it for review.
- **Perform server-side authoritative actions** – Cloud Canvas can call your Lambda functions to control in-game economies. For example, when a player tries to purchase an item, your Lambda function can check a database to verify that the player has enough money to pay for the item. The function can then deduct the amount from the player's account, and add the item to the player’s inventory.

Identity and Permissions

For managing the identity of the player and controlling access to AWS resources in the cloud, Cloud Canvas supports the Amazon Cognito service.

Amazon Cognito can create unique anonymous identities for your users tied to a particular device. It can also authenticate identities from identity providers like Login with Amazon, Facebook, or Google. This provides your game with a consistent user IDs that can seamlessly transition from anonymous use on a single device to authenticated use across multiple devices. Consider these examples:

- Players start playing your game anonymously and store their progress locally on their device. Later, to "upgrade" their experience, you ask them to be authenticated through one of the login providers mentioned. After players provide an authenticated ID, you can store their progress in the cloud, and they can access their progress across multiple devices.
- You can specify which AWS resources players are allowed to access. For example, you can enable the "Get the Latest High Scores" Lambda function to be called not only by your game, but by anyone, including external websites. But you could specify that the "Submit High Scores" function only be called by players of your game so that your high score table remains secure. You can use Cloud Canvas to manage these permissions.

Cloud Canvas Resource Management

In addition to communicating with Amazon Web Services, Cloud Canvas can also help you manage your resources. Amazon Web Services can help create and manage any cloud resources that a game feature needs. Once you implement the feature you can use Cloud Canvas deployments to manage the resources for development, test, and live versions of your game.

Defining the Resources

You can create cloud resources by using AWS CloudFormation templates. AWS CloudFormation is an Amazon Web Service with which you can define, create, and manage AWS resources predictably and repeatedly by using templates. The templates are JSON-formatted text files that you use to specify the collection of resources that you want to create together as a single unit (a stack).

In a template, each resource gets its own AWS CloudFormation definition in which you specify the parameters that govern the resource. AWS CloudFormation templates are beyond the scope of this topic, but for now it’s enough to understand that you can define (for example) a template with an Amazon DynamoDB table and two AWS Lambda functions. For an example AWS CloudFormation template that creates an Amazon DynamoDB table, see the AWS CloudFormation User Guide.

Deployments

While you are working on a new feature, your quality assurance team might have to test it. You want to provide a version of your feature that the test team can use while you continue to work on your own version. To keep the corresponding resources of the different versions distinct, Cloud Canvas gives you the ability to create separate deployments. Deployments are distinct instances of your product’s features.

In a scenario like the one described, you might create three deployments: one for the development team, one for the test team, and one for live players. Each deployment's resources are independent of each other and can contain different data because (for example) you don’t want the data entered by the test team to be visible to players.

With Cloud Canvas you can manage each of these deployments independently of one another, and you can switch between deployments at will. After making changes, you can use Cloud Canvas to update your feature or deployment and update the corresponding AWS resources.

Team Workflow Using Deployments

The following workflow example illustrates how Cloud Canvas deployments work:

1. The test team finds a bug. You fix the bug in your Lambda code.
2. You switch to the dev deployment and upload the new version of the Lambda function. The Lambda code in the test and live deployments remain untouched for now, and they continue working as is.
3. After you are satisfied that the bug has been fixed, you update the Lambda code in the test deployment. The test team can now test your fix. The live deployment continues unchanged.
4. After the test team approves the fix, you update the live deployment, propagating the fix to your live players without requiring them to download a new version of the game.

Communicating with Cloud Resources using Flow Graph

As your game communicates with its AWS resources, you can use Lumberyard's flow graph system to implement the interaction between your game and AWS. Cloud Canvas-specific flow graph nodes function just like other flow graph nodes, but they make calls to AWS services. For example, if your feature uses two Lambda functions that are needed in different situations, you can use the Lumberyard flow graph system to specify that the functions get called under the appropriate conditions in your game.

You can also use flow graph to take appropriate actions depending on the success or failure of a function. For example, your function might return failure when no Internet connection exists, or when the function lacks sufficient permissions to contact the resource. Your game can parse any failures and handle them appropriately, such as asking the user to retry or skip retrying.

When you have multiple deployments, Cloud Canvas keeps an internal mapping of friendly names to AWS instances so that your game knows which AWS resources to use. Cloud Canvas maps the currently selected deployment to the corresponding set of resources.

Thus, when you release your game to customers, you use a deployment specifically set aside for live players. If you are using the dev version of one feature and switch your deployment to test, your game calls the Lambda function associated with the test deployment.

Managing Permissions Using Cloud Canvas

Managing permissions is an important part of building a secure cloud-connected game. Maintaining separate and distinct permissions is important in the phases of development, testing, and production. You can apply permissions to your development and test teams, to the AWS resources that your game uses, and to the players of your game. A key objective is to secure your game's AWS resources against hackers and other forms of abuse.

You can use permissions to specify exactly who is allowed to do what to the AWS resources that are part of your game. For example, if you have a game feature that uploads screenshots, you can create an Amazon S3 bucket to store the screenshots. You can set permissions for the game to be able to write (send files) to the bucket, but not read from the bucket. This prevents inquisitive users from examining the files that have been uploaded. On the other hand, you can give your team members permissions to read files from the bucket so that they can review and approve them. With Cloud Canvas you can also set the permissions for individual deployments. For example, live and test deployments can have different permission sets.

Like features, you can define permissions through AWS CloudFormation templates. The permissions are applied any time that you update your cloud resources using the Cloud Canvas resource management tools.

For more information, see Access Control and Player Identity (p. 187).

Tutorial: Initializing and Administering a Cloud Canvas Project

This tutorial walks you through the steps of getting started with Cloud Canvas, including signing up for an Amazon Web Services (AWS) account, providing your AWS credentials, and using the command line tools to initialize Cloud Canvas. At the end of the tutorial you will have used your AWS credentials to administer a Cloud Canvas-enabled Lumberyard project.

Specifically, this tutorial guides you through the following tasks:

- Obtain an Amazon Web Services account.
- Navigate the AWS Management Console.
- Create an AWS Identity and Access Management (IAM) user with suitable permissions to administer a Cloud Canvas project.
- Get credentials from your IAM user and type them into the Cloud Canvas tools.
- Use the command line tool to initialize a Lumberyard project for use with Cloud Canvas.
- Dismantle the project, removing all AWS resources that were allocated by Cloud Canvas.

Prerequisites

Before starting this tutorial, you must complete the following:

- Install a working version of Lumberyard Editor.
- Set up a Lumberyard project with the Cloud Canvas Gem (extension) enabled.
- Read through the Cloud Canvas introduction and Cloud Canvas concepts.

Step 1: Sign up for AWS

When you sign up for Amazon Web Services (AWS), you can access all its cloud features. Cloud Canvas creates resources in your AWS account to make these services accessible through Lumberyard. You are charged only for the services that you use. If you are a new AWS customer, you can get started with Cloud Canvas for free. For more information, see AWS Free Tier.

If you or your team already have an AWS account, skip to Step 2 (p. 132).

To create an AWS account

1. Open https://aws.amazon.com/ and then choose **Create an AWS Account**.
2. Follow the online instructions to create a new account.

   Note

   - As part of the sign-up procedure, you will receive a phone call and enter a PIN using your phone.
   - You must provide a payment method in order to create your account. Although the tutorials here fall within the AWS Free Tier, be aware that you can incur costs.

3. Wait until you receive confirmation that your account has been created before proceeding to the next step.
4. Make a note of your AWS account number, which you will use in the next step.

You now have an AWS account. Be sure to have your AWS account number ready.

Step 2: Create an AWS Identity and Access Management (IAM) User for Administering the Cloud Canvas Project

After you confirm that you have an AWS account, you need an AWS Identity and Access Management (IAM) user with adequate permissions to administer a Cloud Canvas project. IAM allows you to manage access to your AWS account.

AWS services require that you provide credentials when you access them to verify that you have the appropriate permissions to use them. You type these credentials into Lumberyard Editor as part of setting up your project.

The IAM user that you will create will belong to a group that has administrator permissions to install the Cloud Canvas resources and make them accessible through Lumberyard. Administrative users in this group will have special permissions beyond the scope of a normal Cloud Canvas user.

In a team environment, you—as a member of the administrator's group—can create IAM users for each member of your team. With IAM you can set permissions specifically for each person’s role in a project. For example, you might specify that only designers may edit a database, or prevent team members from accidentally writing to resources with which your players interact.

For more information on IAM and permissions, see the IAM User Guide.

This section guides you through IAM best practices by creating an IAM user in your account, creating an administrator group, and then adding the IAM user to the administrator group.

Create an IAM user

It's time to create your IAM user.

To create an IAM user in your account

1. Sign into the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, click **Users**.
3. Click **Create New Users**.
4. Type a user name into box **1**. This tutorial uses the name *`CloudCanvasAdmin`*.
5. Ensure that the **Generate an access key for each user** check box is checked.
6. Click **Create**. Your IAM user is created along with two important credentials: an access key and a secret access key. These credentials must be entered into Cloud Canvas in order to access the AWS resources in your project.
7. Click **Show User Security Credentials** to view your security credentials, or click **Download Credentials** to download them in a `.csv` file. Make sure you preserve the credentials in a safe place now. After this point, you cannot view the secret access key from the AWS Management Console.

   Important
   Do not share your credentials with anyone. Anyone with access to these credentials can access your AWS account, incur charges, or perform malicious acts.
8. Click **Close**. Next, you create a password for the user.
9. In the list of users, click the *`CloudCanvasAdmin`* user name (or whatever name you used).
10. Click the **Security Credentials** tab to create a password so that you can sign in as this user later.
11. Click **Manage Password**.
12. Click **Assign an auto-generated password** or **Assign a custom password**.
13. When you are finished, choose **Apply**.

    You have now created an IAM user. Next, you create an administrator group to which you later add the *`CloudCanvasAdmin`* user.

Create a Group for Administrators

IAM groups simplify managing permissions for a number of users without having to manage each user individually.

To create a group for your Cloud Canvas administrative IAM user

1. In the left navigation pane of the IAM console, choose **Groups**.
2. Choose **Create New Group**.
3. Give the group a name, such as *`CloudCanvasAdministrators`*.
4. Click **Next Step**.
5. Select the check box next to the **AdministratorAccess** policy. This policy provides the necessary permissions for creating and administering a Cloud Canvas project.

Warning
The **AdministratorAccess** policy allows almost all permissions within the AWS account and should be attached only to the administrator of the account. Otherwise, other team members could perform actions that incur unwanted charges in your AWS account.

6. Click **Next Step**.
7. Review the proposed group that you are about to create.
8. Click **Create Group**.

You have now created a group with administrator permissions.

Next, you add the *CloudCanvasAdmin* IAM user that you created earlier to the *CloudCanvasAdministrators* group that you just created.

Add an IAM User to an Administrator Group

Adding users to an administrator group is easier and more secure than attaching a policy with administrator permissions separately to each IAM user. In this tutorial, you add only one IAM user to the administrator group, but you can add more if required.

To add the *CloudCanvasAdmin* IAM user to the *CloudCanvasAdministrators* group

1. Click the name of the newly created *CloudCanvasAdministrators* group (not the check box).
2. In the **Users** tab, choose **Add Users to Group**.
3. Select the check box next to the *CloudCanvasAdmin* IAM user that you want to belong to the administrator group.
4. Click **Add Users**. The *CloudCanvasAdmin* user name now appears in the list of users for the *CloudCanvasAdministrators* group.
5. Sign out of the AWS Management Console so that you can sign in as your *CloudCanvasAdmin* user in the next step.

Step 3: Sign in as Your IAM User

Now you're ready to try out your new user.

To sign in as your IAM user

1. Get the AWS account ID that you received when you created your AWS account. To sign in as your *CloudCanvasAdmin* IAM user, use this AWS account ID.
2. In a web browser, type the URL `https://<your_aws_account_id>.signin.aws.amazon.com/console/`, where *<your_aws_account_id>* is your AWS account number without the hyphens. For example, if your AWS account number is *1234-5678-9012*, your AWS account ID would be *123456789012*, and you would visit `https://123456789012.signin.aws.amazon.com/console/` .

   For convenience, you might want to bookmark your URL for future use.
3. Type the *CloudCanvasAdmin* IAM user name you created earlier.
4. Type the password for the user and choose **Sign In**.

You are now successfully signed into the AWS Management Console.

Note
Throughout the tutorial, you must be signed into the AWS Management Console. If you are signed out, follow the preceding steps to sign back in.

Step 4: Enabling the Cloud Canvas Gem (extension) Package

Cloud Canvas functionality is enabled in Lumberyard through a Gem package. Gem packages, or Gems, are extensions that share code and assets among Lumberyard projects. You access and manage Gems through the Project Configurator. The default SamplesProject is already configured to use the Cloud Canvas Gem package.

This section of the tutorial shows you how to use the SamplesProject, and how to enable the Cloud Canvas Gem package in a new project if you are not using the SamplesProject.

Enable Cloud Canvas in the SamplesProject

If you are using the SamplesProject , follow these steps to enable Cloud Canvas functionality.

To use the SamplesProject

1. Launch `ProjectConfigurator.exe` from your Lumberyard `dev\Bin64\` binary directory.
2. Under **SamplesProject**, click **Launch editor**.
3. Go to Step 5: Add Administrator Credentials to Lumberyard (p. 135).

Enable Cloud Canvas in a New Project

If you are working on a new project, follow these steps to enable Cloud Canvas functionality.

> **Note**
> Adding the Cloud Canvas Gem package to a project that is not already configured requires rebuilding the project in Visual Studio.

To enable Cloud Canvas in a new project

1. Launch `ProjectConfigurator.exe` from your Lumberyard `dev\Bin64\` binary directory.
2. Click **Enable packages** to navigate to the Gems packages screen.
3. Ensure that the check box for the **Cloud Canvas (AWS)** Gem package is checked. If it is already checked, close the **ProjectConfigurator** and go to Step 5: Add Administrator Credentials to Lumberyard (p. 135) now.
4. Click **Save** and close the **ProjectConfigurator**.
5. If you had to add the Cloud Canvas (AWS) gem to the project, open a command line window and run **`lmbr_waf configure`** to configure your new project.
6. Recompile and build the resulting Visual Studio solution file. Your Lumberyard project is now ready for Cloud Canvas.

Step 5: Add Administrator Credentials to Lumberyard

In order to begin managing a Cloud Canvas project, you add the IAM user credentials that you generated earlier to a profile that Cloud Canvas can easily reference.

To type your credentials

1. In Lumberyard Editor, click **AWS**, **Cloud Canvas**, **Permissions and deployments**.

2. On the **Authentication** tab, for **Profile** name, type a friendly name, such as *CloudCanvasAdminProfile*. Note this name for later use. This name does *not* have to be the same as the IAM user that you typed in Step 2 (p. 132).
3. Enter the access key and secret access key that you generated in Step 2 (p. 132).
4. Click **Save**, and then click **Close**. The profile name is now associated with your credentials, and saved locally on your machine in your AWS credentials file. This file is normally located in your `C:\Users\`*<user name>*`\.aws\` directory. As a convenience, other tools such as the AWS Command Line Interface or the AWS Toolkit for Visual Studio can access these credentials.

   Important
   Do not share these credentials with anyone, and do not check them into source control. These grant control over your AWS account, and a malicious user could incur charges.

You have now created a profile for administering a Cloud Canvas project.

Step 6: Initializing Cloud Canvas from the Command Line

In this step, you configure your Lumberyard project to use Cloud Canvas features. It sets up all of the initial AWS resources required by Cloud Canvas. You perform this step only once for any project.

To initialize Cloud Canvas

1. If you have checked Lumberyard into source control, ensure that the *<base Lumberyard directory>*`\`*<project name>*`\AWS\project-settings.json` file has been checked out and is writeable. The initialization process writes information related to where the Cloud Canvas AWS resources are deployed to this file. In a team environment, check this file into source control so that other team members have access to the AWS resources.
2. Open a command line window and change to your root Lumberyard directory.
3. You must provide two pieces of information to Cloud Canvas when initializing it: the profile name that supplies the required AWS credentials, and the region to which to deploy AWS resources. You created a profile name in Step 5 (p. 135) (for example, *CloudCanvasAdminProfile*).
4. Cloud Canvas requires selecting a region that is supported by the Amazon Cognito service. You can check the availability of this service by visiting the Region Table. This tutorial deploys resources to US East (N. Virginia), which supports Amazon Cognito.
5. Type the following at the command prompt, and then press **Enter**:

```
lmbr_aws initialize-project --region us-east-1 --profile <profile name from
 Step 5>
```

   For example, if the profile name you selected is *CloudCanvasAdminProfile*, type the following:

```
lmbr_aws initialize-project --region us-east-1 --profile CloudCanvasAdmin
Profile
```

   This command creates resources locally and in your AWS account and copies files for project administration.
6. Wait until the initialization process is complete before you proceed. The initialization process can take several minutes.

   Tip
   The initialization process has to be done only once for a given Lumberyard project.

7. If you are using source control, check in the `settings.json` file so that other users on your team can access the AWS resources.

Your Lumberyard project is now ready to use Cloud Canvas features.

Step 7: Inspecting Your AWS Account

This step in the tutorial shows you the AWS CloudFormation stacks that the initialization process created for you.

To inspect your AWS account

1. In a web browser, use your IAM credentials to sign in to the AWS Management Console (see Step 3 (p. 134)).
2. Ensure the AWS region, which appears on the upper right of the console screen, is set to the one that you specified when you had Cloud Canvas deploy its resources in Step 6 (p. 136). If you selected the region in this tutorial, you will see **N. Virginia**.
3. Click **Services**, **CloudFormation**.
4. Note that a number of other stacks have been created as a result of the initialization process. If the initialization process (Step 6 (p. 136)) is still under way, some will show the status **UPDATE_IN_PROGRESS**. Otherwise, the status shows **CREATE_COMPLETE**. You may need to click **Refresh** to update the status.

The next step shows how, as an administrator, you can grant your team members access to Cloud Canvas features.

Step 8: Using IAM to Administer a Cloud Canvas Team

In this step, you create Cloud Canvas IAM users for your team, create a group for your users, attach a Cloud Canvas managed policy to the group, and then add the users to the group. This helps you manage your users' access to AWS resources.

The policies that Cloud Canvas creates for your IAM users are much more restrictive than those for an administrator. This is so that your team members don't inadvertently incur charges without administrator approval.

As you add new features and AWS resources to your project, Cloud Canvas automatically updates these managed policies to reflect the updated permissions.

Create IAM users

You start by creating one or more IAM users.

To create IAM users

1. Sign in to the AWS Management Console using your *`CloudCanvasAdmin`* credentials (see Step 3 (p. 134)).
2. Click **Services**, **IAM**.
3. In the navigation pane, click **Users**.
4. Click **Create New Users**.
5. Type IAM user names for each team member.

6. Be sure that the **Generate an access key for each user** checkbox is checked.
7. Click **Create**.
8. Choose the option to download the access key and secret access key for each user. The keys for all users that you created are downloaded in a single `.csv` file. Make sure you preserve the credentials in a safe place now. After this point, you cannot view the secret access key from the AWS Management Console. You must deliver each user his or her keys securely.
9. Click **Close**.

Create a group

Next, you create an IAM group for the newly created users.

To create a group for the Cloud Canvas IAM users

1. In the left navigation pane of the IAM console, click **Groups**.
2. Click **Create New Group**.
3. Give the group a name. This tutorial uses the name *`CloudCanvasDevelopers`*.
4. Click **Next Step**.
5. To find the IAM managed policy that Cloud Canvas created for you, click the link next to **Filter** and click **Customer Managed Policies**.
6. Select the check box next to the policy that includes your project name. If you are using the SamplesProject, the name begins with **SamplesProject-DontDieDeploymentAccess**.
7. Click **Next Step**.
8. Review the proposed group that you are about to create.
9. Click **Create Group**.

Add IAM users to a group

Finally, you add your IAM users to the group you just created.

To add your Cloud Canvas IAM users to the group

1. If it is not already selected, click **Groups** in the left navigation pane.
2. Click the name of the newly created *`CloudCanvasDevelopers`* group (not the check box adjacent to it).
3. If it is not already active, click the **Users** tab.
4. Choose **Add Users to Group**.
5. Select the check boxes next to the IAM users that you want to belong to the *`CloudCanvasDevelopers`* group.
6. Click **Add Users**. The team's user names now appear in the list of users for the group.
7. Open the `credentials.csv` file that you downloaded earlier. Securely deliver the secret and access keys to each user in the group. Stress the importance to each user of keeping the keys secure and not sharing them.
8. Have each user in the *`CloudCanvasDevelopers`* group perform the following steps:

    a. In Lumberyard Editor, click **AWS**, **Cloud Canvas**, **Permissions and Deployments**.
    b. Type a new profile name and his or her access and secret access keys.

Important
As an administrator, it is your responsibility to keep your team and your AWS account secure. Amazon provides some best practices and options for how to manage your team's access keys on the Managing Access Keys for IAM Users page. You are encouraged to read this thoroughly.

For information regarding limits on the number of groups and users in an AWS account, see Limitations on IAM Entities and Objects in the IAM User Guide.

Step 9: Use the Command Line to Remove Cloud Canvas Functionality and AWS Resources

This step shows you how to remove Cloud Canvas functionality from your Lumberyard project and remove all AWS resources related to it. A number of safeguards ensure that a team member does not do this accidentally. Only administrators should perform these actions.

Warning
These steps remove all AWS resources managed by Cloud Canvas. After the steps are complete, the players of your game will not be able to access any Cloud Canvas features.

To use the command line to remove Cloud Canvas functionality and release all AWS resources

1. If you have checked Lumberyard into source control, ensure that the *`<base Lumberyard directory>`*`\`*`<project name>`*`\AWS\project-settings.json` has been checked out and is writeable.
2. Open the `project-settings.json` file in a text editor.
3. Find the long string associated with `StackId`, such as `arn:aws:cloudformation:us-east-1:`*`<xxxxx>`*. As a safeguard, Cloud Canvas requires you to remove this value so that you are very intentionally making this step. Select the entire *`arn:aws:cloudformation`* string (not including the quotes) and use the cut feature of your text editor. The line should now read `"StackId" : ""`.
4. Save the file.
5. Open a command line window and change to your root Lumberyard directory.
6. Type the following command, paste the string you cut earlier for *`<stack ID>`*, and press **Enter**.

   Although any errors that occur will be reported, this step is not reversible.

   ```
   lmbr_aws delete-project-stack --project-stack-id <stack ID>
   ```

You have now removed all of the AWS resources that are related to your Cloud Canvas project.

Getting Started

A Lumberyard project that uses AWS requires three things:

1. A *definition* of the AWS resources that the game requires to operate. These definitions are organized into one or more unrelated features like HighScores or SavedGames. For information, see Resource Definitions (p. 143).
2. A *deployment* of those resources in an AWS account. You can perform separate deployments of a game's resources for targeted purposes such as development, test, and release. For more information, see Resource Deployments (p. 166).

3. A *mapping* from the friendly resource names to the actual names of the deployed resources. For example, the friendly name `DailyGiftTable` of a DynamoDB table might get mapped to a name like `SamplesProject-DontDieDeployment-78AIXR0N0O4N-DontDieAWS-1I1ZC6YO7KU7F-DailyGiftTable-1G4G33K16D8ZS`. For more information, see Resource Mappings (p. 168).

Initializing Resource Management for a Project

The first step is to initialize resource management support for a project. This only needs to be done once for a project and must be done by an AWS account administrator. The results of this initialization process should be shared with your team by checking it into source control.

To initialize project resource management

1. Open the AWS Management Console, log into your AWS account (create one if necessary), and retrieve your root credentials as described under **Access keys (access key ID and secret access key)** at How Do I Get Security Credentials?
2. Open a Windows command prompt and type the following:

   ```
   cd {root}
   ```

   Replace *{root}* with the Lumberyard install directory.
3. Ensure that the *{root}*`\bootstrap.cfg` file's `sys_game_folder` property identifies the project that you want to initialize. You can change the active project using the Lumberyard project configuration tool.
4. To initialize the project, type the following:

   ```
   lmbr_aws initialize-project --aws-access-key {access-key} --aws-secret-key
    {secret-key} --region{region}
   ```

   Replace *{access-key}* and *{secret-key}* with the credentials retrieved in the first step and *{region}* with the AWS region that you will use when you develop your project. See AWS Regions and Endpoints for a complete list of available regions. You need to pick a region that supports both Amazon Cognito and AWS CloudFormation.

This command creates some files in the *{root}*`\`*{game}*`\AWS` directory and then prompts you to confirm that you want to use those files to create AWS resources. The default configuration creates a Development deployment with a simple HelloWorld feature that uses a DynamoDB table and a Lambda function resource. To confirm which AWS services are being used, you can view the file contents in a text editor.

The command then creates an AWS CloudFormation stack for your project using the contents of the AWS directory.

At this point, the AWS resources defined by the **HelloWorld** feature have been created and are ready for use. Next, you will use the resources in a game.

To configure resources for a game

1. Start Lumberyard Editor and open a level in your project.
2. Create or open a flow graph attached to an entity in that level. To create a flow graph in Lumberyard Editor, right click the entity and then choose **Create Flow Graph**. To edit an existing flow graph, right click the entity, choose **Flow Graphs**, and select the flow graph that you created.
3. Add the **Game:Start**, **AWS:Configuration:ConfigureAnonymousPlayer**, **AWS:Primitive:Lambda::Invoke** and **Debug::Log** nodes to the graph:

   a. Link the **Game:Start** node's output port to the **ConfigureAnonymousPayer** node's **Configure** port.
   b. Link the **ConfigureAnonymousPlayer** node's **Success** port to the **Lambda:Invoke** node's **Invoke** port.
   c. Set the **Lambda:Invoke** node's **FunctionName** port to **HelloWorld.SayHello**.
   d. Link the **Lambda:Invoke** node's **Success** port to the **Debug:Log** node's input port.
   e. Link the **Lambda:Invoke** node's **Result** port to the **Debug:Log** node's message port.

4. Start the game by pressing **Ctrl+G**.
5. The console in the editor should show a **HelloWorld.SayHello -> Hello World!** message.
6. Finally, add the contents of the *{root}\{game}*`\AWS directory` (including all subdirectories) to your source control system.

Modifying and Updating a Feature

The AWS resources that a project uses are organized into features. A project may define any number of features, each with their own independent resources.

The default configuration defines a HelloWorld feature that you can delete or modify as needed.

To modify a feature

1. Make changes to the feature's `feature-template.json` file or the contents of `feature-code`.
2. Type the following

```
lmbr_aws update-feature --feature-name {feature-name} --deployment-name
{deployment-name}
```

   Replace *{feature-name}* and *{deployment-name}* with the name of the feature and deployment that you want to update, respectively.

   The following example command updates the **HelloWorld** feature of the **Development** deployment that was created when the project was initialized.

```
lmbr_aws update-feature --feature-name HelloWorld --deployment-name Develop
ment
```

Adding a Feature and Updating a Deployment

To add additional features

Now you're ready to add a feature.

1. To add a feature, type:

```
lmbr_aws add-feature --feature-name {feature-name}
```

   Replace *{feature-name}* with the name of the feature that you want to add. This command adds a `FeatureConfiguration` and AWS CloudFormation stack resources to your `deployment-template.json` file. These are similar to the HelloWorldConfiguration Resource (p. 154) and the HelloWorld Resource (p. 154) resource in the example deployment-template.json (p. 152) file. The command also creates a *`{game}`*`\feature\`*`{feature-name}`* directory with a default `feature-template.json` file and `feature-code` subdirectory.

2. Edit the contents of the *`{game}`*`\feature\`*`{feature-name}`* directory to define your feature's resources.
3. To continue developing and testing the new feature, type the following to update the deployment.

```
lmbr_aws update-deployment --deployment-name {deployment-name}
```

   Replace *`{deployment-name}`* with the name of the deployment that you want to update.
4. To add the new feature to all the project's deployments, type the following:

```
lmbr_aws update-project
```

5. Create game code and/or flow graphs to interact with the resources defined by your new feature. Use names like *`{feature}.{resource}`* in flow nodes to identify the resources in your feature. In C++ code, you can use these names to retrieve a resource client from the client configuration system (for more information, see Mappings Property (p. 146)).

Adding a Deployment and Updating the Project

Each of a project's deployments embodies a complete and fully independent copy of all the game's resources. A project may define any number of deployments. You may choose to create Development, Test, and Release deployments, or create deployments for each of your feature teams.

The default configuration defines a **Development** deployment, which you can delete or modify as needed.

Note
Only project administrators (anyone with full AWS account permissions) can add and remove deployments.

To add additional deployments

1. Type the following:

```
lmbr_aws add-deployment --deployment-name {deployment-name}
```

   Replace *`{deployment-name}`* with the name of the deployment that you want to add. This command adds `DeploymentConfiguration` and AWS CloudFormation stack resources to your `project-template.json` file. These resources will be similar to the DevelopmentConfiguration Resource (p. 152), Development Resource (p. 152), and DevelopmentAccess Resource (p. 152) that are in the example project-template.json (p. 146) file.
2. To create the resources for the new deployment, update the project with the following command.

```
lmbr_aws update-project
```

Resource Definitions

Resource definitions are specifications in AWS CloudFormation templates that determine the resources (for example, DynamoDB databases, Lambda functions, and access control information) that will be created in AWS for the game. Game code and flow graphs use AWS resources and expect those resources to exist and to be configured in a specific way. The resource definitions determine this architecture and configuration.

Resource Definition Location

A description of the resources required by the game is stored under the *{root}\{game}*\AWS directory, where *{root}* is the Lumberyard install directory and *{game}* is the directory identified by the `sys_game_folder` property in the *{root}*\bootstrap.cfg file. These definitions should be checked into the project's source control system along with all your other game code and data.

The default *{game}*\AWS directory contents are created by the `lmbr_aws` initialize-project (p. 174) command.

In addition, some user-specific configuration data is kept in the *{root}*\Cache\*{game}*\pc\user\AWS directory. The contents of this directory should not be checked into the project's source control system.

The following shows the contents of these `AWS` directories.

```
{root}\{game}\AWS\
    project-settings.json
    project-template.json
    deployment-template.json
    deployment-access-template.json
    project-code\
        (Lambda function Code)
    feature\
        {feature}\
            feature-template.json
            feature-code\
                (Lambda function Code)

{root}\Cache\{game}\pc\user\AWS\
    user-settings.json
```

Each of these `.json` files is described in the following sections.

project-settings.json

The `project-settings.json` file contains project configuration data. The structure of this file is as follows:

```
{
    "{key}": "{value}",
    "deployment": {
        "{deployment}": {
            "{key}": "{value}",
            "feature": {
                "{feature}": {
                   "{key}": "{value}"
                }
            }
        }
    }
}
```

The *{key}* and *{value}* pairs represent individual settings. The pairs at the root apply to the project. The pairs under *{deployment}* apply to that deployment. The pairs under *{feature}* apply to that feature. Either or both of *{deployment}* and *{feature}* can be `*`, to indicate the settings they contain apply to all deployments or features, respectively. Settings under a named entry take precedence over settings under a `*` entry.

An example `project-settings.json` settings file follows.

```
{
    "StackId": "arn:aws:cloudformation:us-west-2:xxxxxxxxxxxx:stack/My
Game/yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy",
    "DefaultDeployment": "Development",
    "ReleaseDeployment": "Release",
    "deployment": {
        "*": {
            "feature": {
                "HelloWorld": {
                   "parameter": {
                        "WriteCapacityUnits": 1,
                        "ReadCapacityUnits": 1,
                        "Greeting": "Hi"
                    }
                }
            }
        },
        "Development": {
           "feature": {
                "HelloWorld": {
                    "parameter": {
                        "WriteCapacityUnits": 5,
                        "ReadCapacityUnits": 5
                    }
                }
            }
        }
    }
}
```

StackId Property

The `StackId` property identifies the AWS CloudFormation stack for the project. The project's deployment stacks are children of this stack. For more information, see Resource Deployments (p. 166).

The `StackId` property is set by the initialize-project (p. 174) command. If for some reason you want to associate the project with an existing project stack, you can use the AWS Management Console to look up the stack's ARN and paste it into the `project-settings.json` file (navigate to AWS CloudFormation, select the stack, select **Overview**, and then copy the value of the `Stack Id` property).

DefaultDeployment Property

The `DefaultDeployment` property identifies the deployment that is to be used by default when working in Lumberyard Editor. The `DefaultDeployment` property in the `user-settings.json (p. 145)` file overrides this setting. The project and user defaults can be set using the `lmbr_aws` default-deployment (p. 173) command. The `DefaultDeployment` setting is also used by the `lmbr_aws` update-mappings (p. 181) command.

ReleaseDeployment Property

The `ReleaseDeployment` property identifies the deployment that is to be used in release builds of the game. The `ReleaseDeployment` setting is used by the `lmbr_aws` update-mappings (p. 181) command.

parameter Property

The `parameter` property provides the values for feature template parameters. The property must be in the following format.

```
{
    ...
                    "parameter": {
                        "{template-parameter-name-1}": {template-parameter-
value-1},
                        ...
                        "{template-parameter-name-n}": {template-parameter-
value-n}
                    }
    ...
}
```

user-settings.json

The `user-settings.json` file contains user-specific configuration data.

File Location

The `user-settings.json` file is found at *{root}*`\Cache\`*{game}*`\pc\user\AWS\user-settings.json`. It is not in the *{root}*`\`*{game}*`\AWS` directory along with the other files described in this section because it should not be checked into the project's source control system.

An example `user-settings.json` file follows.

```
{
    "DefaultDeployment": "Test",
    "Mappings": {
        "HelloWorld.SayHello": {
            "ResourceType": "AWS::Lambda::Function",
            "PhysicalResourceId": "MyGame-Test-xxxxxxxxxxxxx-HelloWorld-
yyyyyyyyyyyy-SayHello-zzzzzzzzzzzz"
        }
    }
}
```

DefaultDeployment Property

The `DefaultDeployment` property identifies the deployment that is to be used by default when working in Lumberyard Editor. The `DefaultDeployment` property in the `user-settings.json` file overrides the property from the **`project-settings.json`** (p. 143) file. The project and user defaults can be set using the `lmbr_aws` default-deployment (p. 173) command.

Mappings Property

The `Mappings` property specifies the mapping of friendly names used in Lumberyard Editor to actual resource names. For example, the **DailyGiftTable** DynamoDB table would get mapped to a name like `SamplesProject-DontDieDeployment-78AIXR0N0O4N-DontDieAWS-1I1ZC6YO7KU7F-DailyGiftTable-1G4G33K16D8ZS`.

This property is updated automatically when the default deployment changes or when the default deployment is updated. It can be refreshed manually by using the `lmbr_aws` update-mappings (p. 181) command.

project-template.json

The `project-template.json` file is an AWS CloudFormation template that defines a child AWS CloudFormation stack resource for each of the project's deployments as well as some resources that support the Cloud Canvas resource management system.

An example `project-template.json` file follows.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Parameters": {
        "ConfigurationKey": {
            "Type": "String",
          "Description": "Location in the configuration bucket of configuration
 data."
        }
    },
    "Resources": {
        "Configuration": {
            "Type": "AWS::S3::Bucket",
            "DeletionPolicy": "Retain",
            "Properties": {
                "VersioningConfiguration": {
                    "Status": "Enabled"
                },
                "LifecycleConfiguration": {
                    "Rules": [
                        {
                            "Id": "DeleteOldVersions",
                            "NoncurrentVersionExpirationInDays": "2",
                            "Status": "Enabled"
                        },
                        {
                            "Id": "DeleteUploads",
                            "Prefix": "uploads",
                            "ExpirationInDays": 2,
                            "Status": "Enabled"
                        }
                    ]
                }
            }
        },
        "ProjectPlayerAccessTokenExchangeHandlerRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Action": "sts:AssumeRole",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            }
                        }
                    ]
                },
                "Policies": [
                    {
                        "PolicyName": "PlayerAccessTokenExchange",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Sid": "WriteLogs",
                                    "Effect": "Allow",
                                    "Action": [
                                        "logs:CreateLogGroup",
                                        "logs:CreateLogStream",
                                        "logs:PutLogEvents"
                                    ],
                                    "Resource": "arn:aws:logs:*:*:*"
                                },
                                {
                                    "Sid": "GetAuthSettings",
                                    "Action": [
                                        "s3:GetObject",
                                        "s3:HeadObject"
                                    ],
                                    "Effect": "Allow",
                                    "Resource": [
                                        { "Fn::Join": [ "", [
                                            "arn:aws:s3:::",
                                            { "Ref": "Configuration" },
                                           "/player-access/auth-settings.json"

                                        ]] }
                                    ]
                                },
                                {
                                    "Sid": "DescribeStacks",
                                    "Action": [
                                     "Cloud Formation:DescribeStackResources",

                                       "Cloud Formation:DescribeStackResource"

                                    ],
                                    "Effect": "Allow",
                                    "Resource": [
                                        "*"
                                    ]
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "ProjectResourceHandlerExecution": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Action": "sts:AssumeRole",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            }
                        }
                    ]
                },
                "Policies": [
                    {
                        "PolicyName": "ProjectAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Sid": "WriteLogs",
                                    "Effect": "Allow",
                                    "Action": [
                                        "logs:CreateLogGroup",
                                        "logs:CreateLogStream",
                                        "logs:PutLogEvents"
                                    ],
                                    "Resource": "arn:aws:logs:*:*:*"
                                },
                                {
                                   "Sid": "ReadAndWriteUploadedConfiguration",

                                    "Effect": "Allow",
                                    "Action": [
                                        "s3:GetObject",
                                        "s3:PutObject"
                                    ],
                                    "Resource": { "Fn::Join": [ "", [
                                        "arn:aws:s3:::",
                                        { "Ref": "Configuration" },
                                        "/upload/*"
                                    ]] }
                                },
                                {
                                    "Sid": "DescribeStackResources",
                                    "Effect": "Allow",
                                    "Action": [
                                     "Cloud Formation:DescribeStackResources",

                                       "Cloud Formation:DescribeStackResource"

                                    ],
                                    "Resource": [
                                        "*"
                                    ]
                                },
                                {
                                    "Sid": "ManagePlayerAndFunctionRoles",
                                    "Effect": "Allow",
                                    "Action": [
                                        "iam:CreateRole",
                                        "iam:DeleteRole",
                                        "iam:GetRole",
                                        "iam:DeleteRolePolicy",
                                        "iam:PutRolePolicy"
                                    ],
                                    "Resource": { "Fn::Join": [ "", [
                                        "arn:aws:iam::",
                                        {"Ref": "AWS::AccountId"},
                                        ":role/",
                                        {"Ref": "AWS::StackName"},
                                        "/*"
                                    ]] }
                                },
                                {
                                    "Sid": "CreateUpdateCognitoIdentity",
                                    "Effect": "Allow",
                                    "Action": [
                                        "cognito-identity:*"
                                    ],
                                    "Resource": { "Fn::Join": [ "", [
                                        "arn:aws:cognito-identity:",
                                        {"Ref": "AWS::Region" },
                                        ":",
                                        { "Ref": "AWS::AccountId" },
                                        ":identitypool/*"
                                    ]] }
                                },
                                {
                                    "Sid": "ReadPlayerAccessConfiguration",
                                    "Effect": "Allow",
                                    "Action": [
                                        "s3:GetObject"
                                    ],
                                    "Resource": { "Fn::Join": [ "", [
                                        "arn:aws:s3:::",
                                        { "Ref": "Configuration" },
                                        "/player-access/auth-settings.json"
                                    ]] }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "ProjectResourceHandler": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Implements the custom resources used in this
project's templates.",
                "Handler": "custom_resource.handler",
                "Role": { "Fn::GetAtt": [ "ProjectResourceHandlerExecution",
"Arn" ] },
                "Runtime": "python2.7",
                "Timeout" : 90,
                "Code": {
                    "S3Bucket": { "Ref": "Configuration" },
                   "S3Key": { "Fn::Join": [ "/", [ { "Ref": "ConfigurationKey"
 }, "project-code.zip" ]] }
                }
            }
        },
        "ProjectPlayerAccessTokenExchangeHandler": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Implements the token exchange for oAuth and
openid used for player access.",
                "Handler": "auth_token_exchange.handler",
              "Role": { "Fn::GetAtt": [ "ProjectPlayerAccessTokenExchangeHand
lerRole", "Arn" ] },
                "Runtime": "python2.7",
                "Code": {
                    "S3Bucket": { "Ref": "Configuration" },
                   "S3Key": { "Fn::Join": [ "/", [ { "Ref": "ConfigurationKey"
 }, "project-code.zip" ]] }
                }
            }
        },
        "DevelopmentConfiguration": {
            "Type": "Custom::DeploymentConfiguration",
            "Properties": {
                "ServiceToken": { "Fn::GetAtt": [ "ProjectResourceHandler",
"Arn" ] },
                "ConfigurationBucket": { "Ref": "Configuration" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "DeploymentName": "Development"
            }
        },
        "Development": {
            "Type": "AWS::Cloud Formation::Stack",
            "Properties": {
                "TemplateURL": { "Fn::GetAtt": [ "DevelopmentConfiguration",
"DeploymentTemplateURL" ] },
                "Parameters": {
                 "ProjectResourceHandler": { "Fn::GetAtt": [ "ProjectResource
Handler", "Arn" ] },
                  "ConfigurationBucket": { "Fn::GetAtt": [ "DevelopmentConfig
uration", "ConfigurationBucket" ] },
                   "ConfigurationKey": { "Fn::GetAtt": [ "DevelopmentConfigur
ation", "ConfigurationKey" ] }
```

```
                }
            }
        },
        "DevelopmentAccess": {
            "Type": "AWS::Cloud Formation::Stack",
            "Properties": {
                "TemplateURL": { "Fn::GetAtt": [ "DevelopmentConfiguration",
"AccessTemplateURL" ] },
                "Parameters": {
                 "ProjectResourceHandler": { "Fn::GetAtt": [ "ProjectResource
Handler", "Arn" ] },
                  "ConfigurationBucket": { "Fn::GetAtt": [ "DevelopmentConfig
uration", "ConfigurationBucket" ] },
                    "ConfigurationKey": { "Ref": "ConfigurationKey" },
                   "ProjectPlayerAccessTokenExchangeHandler":  { "Fn::GetAtt":
  ["ProjectPlayerAccessTokenExchangeHandler", "Arn"] },

                    "ProjectStack": { "Ref": "AWS::StackName" },
                    "DeploymentName": { "Fn::GetAtt": [ "DevelopmentConfigura
tion", "DeploymentName" ] },
                    "DeploymentStack": { "Fn::GetAtt": [ "Development", "Out
puts.StackName" ] },
                    "DeploymentStackArn": { "Ref": "Development" }
                }
            }
        }
    }
}
```

ConfigurationKey Parameter

The `ConfigurationKey` parameter identifies the location of configuration data in the configuration bucket. The parameter value is set by Cloud Canvas when you use the template to update the AWS CloudFormation stack.

Configuration Resource

The `Configuration` resource describes the Amazon S3 bucket that is used to store project configuration data.

ProjectResourceHandlerExecution Resource

The `ProjectResourceHandlerExecution` resource describes the IAM role that is used to grant permissions to the `ProjectResourceHandler` Lambda function resource.

ProjectResourceHandler Resource

The `ProjectResourceHandler` resource describes the Lambda function that implements the AWS CloudFormation custom resource handler that implements the custom resources used in the project's AWS CloudFormation templates. The code for this Lambda function is uploaded from the *{game}*`\AWS\project-code` directory by the `lmbr_aws` initialize-project (p. 174) and update-project (p. 182) commands. For more information, see Custom Resources (p. 183).

ProjectPlayerAccessTokenExchangeHandlerRole Resource

The `ProjectPlayerAccessTokenExchangeHandlerRole` resource describes the IAM role that is used to grant permissions to the `ProjectPlayerAccessTokenExchangeHandler` resource.

ProjectPlayerAccessTokenExchangeHandler Resource

The `ProjectPlayerAccessTokenExchangeHandler` resource describes the Lambda function that implements the token exchange process for player access. The code for this Lambda function is uploaded from the *{game}*`\AWS\project-code` directory by the `lmbr_aws` initialize-project (p. 174) and update-project (p. 182) commands. For more information, see Access Control and Player Identity (p. 187).

DevelopmentConfiguration Resource

The `DevelopmentConfiguration` resource describes a DeploymentConfiguration (p. 184) custom resource that is used to configure the Development resource.

The `project-template.json (p. 146)` contains a similar `DeploymentConfiguration` resource for each of the project's deployments.

Development Resource

The `Development` resource describes the AWS CloudFormation stack that implements the project's Development deployment. The `deployment-template.json (p. 152)` file defines the contents of this stack (and the deployment stack for all other deployments that are added to the project).

The `project-template.json` file contains a similar AWS CloudFormation stack resource for each of the project's deployments.

DevelopmentAccess Resource

The `DevelopmentAccess` resource describes the AWS CloudFormation stack that implements the access controls for the project's Development deployment. The `deployment-access-template.json (p. 154)` file defines the contents of this stack (and the deployment access stack for all other deployments that are added to the project).

The `project-template.json` file contains a similar AWS CloudFormation stack resource for each of the project's deployments.

deployment-template.json

The `deployment-template.json` file is an AWS CloudFormation Template that defines a child AWS CloudFormation stack resource for each of the project's features. As described below, each feature is an arbitrary grouping of the AWS resources used by a game.

An example `deployment-template.json` file follows.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Parameters" : {

        "ProjectResourceHandler": {
            "Type": "String",
            "Description": "Service token of the custom resource handler."
        },
        "ConfigurationBucket": {
            "Type": "String",
            "Description": "Bucket that contains configuration data."
        },
        "ConfigurationKey": {
            "Type": "String",
          "Description": "Location in the configuration bucket of configuration
 data."
        }

    },
    "Resources": {
        "HelloWorldConfiguration" : {
            "Type": "Custom::FeatureConfiguration",
            "Properties": {
                "ServiceToken": { "Ref": "ProjectResourceHandler" },
                "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "FeatureName": "HelloWorld"
            }
        },

        "HelloWorld": {
            "Type": "AWS::Cloud Formation::Stack",
            "Properties": {
                "TemplateURL": { "Fn::GetAtt": [ "HelloWorldConfiguration",
"TemplateURL" ] },
                "Parameters": {
                   "ProjectResourceHandler": { "Ref": "ProjectResourceHandler"
 },
                   "ConfigurationBucket": { "Fn::GetAtt": [ "HelloWorldConfig
uration", "ConfigurationBucket" ] },
                   "ConfigurationKey": { "Fn::GetAtt": [ "HelloWorldConfigura
tion", "ConfigurationKey" ] }
                }
            }
        }
    },
    "Outputs": {
        "StackName": {
            "Description": "The deployment stack name.",
            "Value": {"Ref": "AWS::StackName"}
        }
    }

}
```

Parameters

The `deployment-template.json` file has three parameters:

ProjectResourceHandler Parameter

The `ProjectResourceHandler` parameter identifies the custom resource handler Lambda function used for the project. The parameter value is set by Lumberyard provided tools when you use the template to update the AWS CloudFormation stack.

ConfigurationBucket Parameter

The ConfigurationBucket parameter identifies the configuration bucket. The parameter value is set by Lumberyard provided tools when you use the template to update the AWS CloudFormation stack.

ConfigurationKey Parameter

The ConfigurationKey parameter identifies the location of configuration data in the configuration bucket. The parameter value is set by Lumberyard provided tools when you use the template to update the AWS CloudFormation stack.

Resources

The deployment-template.json file has two resources:

HelloWorldConfiguration Resource

The HelloWorldConfiguration resource describes a FeatureConfiguration (p. 185) custom resource that is used to configure the HelloWorld resource.

The deployment-template.json file contains a similar FeatureConfiguration resource for each of the project's features.

HelloWorld Resource

The HelloWorld resource describes the AWS CloudFormation stack that implements the project's HelloWorld feature.

The deployment-template.json file contains a similar AWS CloudFormation stack resource for each of the project's features.

Outputs

The Outputs section of the template defines values that are consumed in the project-template.json (p. 146) file.

deployment-access-template.json

The deployment-access-template.json file is an AWS CloudFormation Template that defines the resources used to secure a deployment.

An example deployment-access-template.json file follows.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Parameters": {
        "ProjectResourceHandler": {
            "Type": "String",
            "Description": "The project resource handler lambda ARN."
        },
        "ConfigurationBucket": {
            "Type": "String",
            "Description": "Bucket that contains configuration data."
        },
        "ConfigurationKey": {
            "Type": "String",
            "Description": "Key that contains the current upload location."
        },
        "ProjectPlayerAccessTokenExchangeHandler": {
            "Type": "String",
            "Description": "ARN for the lambda that the login Cognito Identity
 pool needs access to."
        },
        "ProjectStack": {
            "Type": "String",
            "Description": "The name of the project stack."
        },
        "DeploymentName": {
            "Type": "String",
            "Description": "The name of the deployment."
        },
        "DeploymentStack": {
            "Type": "String",
            "Description": "The name of the deployment stack."
        },
        "DeploymentStackArn": {
            "Type": "String",
            "Description": "The ARN of the deployment stack."
        }
    },

    "Resources": {
        "OwnerPolicy": {
            "Type": "AWS::IAM::ManagedPolicy",
            "Properties": {
                "Description": "Policy that grants permissions to update a de
ployment stack, and all of it's feature stacks.",
                "Path": { "Fn::Join": [ "", [
                    "/",
                    { "Ref": "ProjectStack" },
                    "/",
                    { "Ref": "DeploymentName" },
                    "/"
                ]] },
                "PolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                           "Sid": "ReadProjectDeploymentAndFeatureStackState",

                            "Effect": "Allow",
                            "Action": [
                                "Cloud Formation:DescribeStackResource",
                                "Cloud Formation:DescribeStackResources",
                                "Cloud Formation:DescribeStackEvents"
                            ],
                            "Resource": [
                                { "Fn::Join": [ "", [
                                    "arn:aws:cloudformation:",
                                    { "Ref": "AWS::Region" },
                                    ":",
                                    { "Ref": "AWS::AccountId" },
                                    ":stack/",
                                    { "Ref": "ProjectStack" },
                                    "/*"
                                ]] },
                                { "Fn::Join": [ "", [
                                    "arn:aws:cloudformation:",
                                    { "Ref": "AWS::Region" },
                                    ":",
                                    { "Ref": "AWS::AccountId" },
                                    ":stack/",
                                    { "Ref": "ProjectStack" },
                                    "-*"
                                ]] }
                            ]
                        },
                        {
                            "Sid": "InvokeProjectResourceHandler",
                            "Effect": "Allow",
                            "Action": [
                                "lambda:InvokeFunction"
                            ],
                            "Resource": [
                                { "Ref": "ProjectResourceHandler" }
                            ]
                        },
                        {
                            "Sid": "ReadAndWriteDeploymentAndFeatureConfigura
tion",
                            "Effect": "Allow",
                            "Action": [
                                "s3:PutObject",
                                "s3:GetObject"
                            ],
                            "Resource": [
                                { "Fn::Join": [ "", [
                                    "arn:aws:s3:::",
                                    { "Ref": "ConfigurationBucket" },
                                    "/upload/*/deployment/",
                                    { "Ref": "DeploymentName" },
                                    "/*"
                                ]] }
                            ]
                        },
                        {
                            "Sid": "UpdateDeploymentStack",
                            "Effect": "Allow",
                            "Action": [
                                "Cloud Formation:UpdateStack"
                            ],
                            "Resource": [
                                { "Fn::Join": [ "", [
                                    "arn:aws:cloudformation:",
                                    { "Ref": "AWS::Region" },
                                    ":",
                                    { "Ref": "AWS::AccountId" },
                                    ":stack/",
                                    { "Ref": "DeploymentStack" },
                                    "/*"
                                ]] }
                            ]
                        },
                        {
                            "Sid": "CreateUpdateAndDeleteFeatureStacks",
                            "Effect": "Allow",
                            "Action": [
                                "Cloud Formation:CreateStack",
                                "Cloud Formation:UpdateStack",
                                "Cloud Formation:DeleteStack"
                            ],
                            "Resource": [
                                { "Fn::Join": [ "", [
                                    "arn:aws:cloudformation:",
                                    { "Ref": "AWS::Region" },
                                    ":",
                                    { "Ref": "AWS::AccountId" },
                                    ":stack/",
                                    { "Ref": "DeploymentStack" },
                                    "-*"
                                ]] }
                            ]
                        },
                        {
                          "Sid": "FullAccessToDeploymentAndFeatureResources",

                            "Effect": "Allow",
                            "Action": [
                                "dynamodb:*",
                                "s3:*",
                                "sqs:*",
                                "sns:*",
                                "lambda:*"
                            ],
                            "Resource": [
                                { "Fn::Join": [ "", [
                                    "arn:aws:dynamodb:",
                                    { "Ref": "AWS::Region" },
                                    ":",
                                    { "Ref": "AWS::AccountId" },
                                    ":table/",
                                    { "Ref": "DeploymentStack" },
                                    "-*"
                                ]] },
                                { "Fn::Join": [ "", [
                                    "arn:aws:s3:::",
                                    { "Ref": "DeploymentStack" },
                                    "-*"
                                ] ] },
                                { "Fn::Join": [ "", [
                                    "arn:aws:sqs:",
                                   { "Ref": "AWS::Region" },
                                   ":",
                                   { "Ref": "AWS::AccountId" },
                                   ":",
                                   { "Ref": "DeploymentStack" },
                                   "-*"
                                ]] },
                                { "Fn::Join": [ "", [
                                    "arn:aws:sns:*:",
                                    { "Ref": "AWS::AccountId" },
                                    ":",
                                    { "Ref": "DeploymentStack" },
                                    "-*"
                                ] ] },
                                { "Fn::Join": [ "", [
                                    "arn:aws:lambda:",
                                    { "Ref": "AWS::Region" },
                                    ":",
                                    { "Ref": "AWS::AccountId" },
                                    ":function:",
                                    { "Ref": "DeploymentStack" },
                                    "-*"
                                ]] }
                            ]
                        },
                        {
                            "Sid": "PassDeploymentRolesToLambdaFunctions",
                            "Effect": "Allow",
                            "Action": [
                                "iam:PassRole"
                            ],
                            "Resource": [
                                { "Fn::Join": [ "", [
                                    "arn:aws:iam::",
                                    {"Ref": "AWS::AccountId"},
                                    ":role/",
                                    {"Ref": "ProjectStack"},
                                    "/",
                                    {"Ref": "DeploymentName"},
                                    "/*"
                                ]] }
                            ]
                        },
                        {
                            "Sid": "CreateLambdaFunctions",
                            "Effect": "Allow",
                            "Action": "lambda:CreateFunction",
                            "Resource": "*"
                        }
                    ]
                }
            }
        },
        "Owner": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "Path": { "Fn::Join": [ "", [
                    "/",
                    { "Ref": "ProjectStack" },
                    "/",
                    { "Ref": "DeploymentName" },
                    "/"
                ]] },
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Sid": "AccountUserAssumeRole",
                            "Effect": "Allow",
                            "Action": "sts:AssumeRole",
                            "Principal": { "AWS": {"Ref": "AWS::AccountId"} }
                        }
                    ]
                },
                "ManagedPolicyArns": [
                    { "Ref": "OwnerPolicy" }
                ]
            }
        },

        "Player": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "Path": { "Fn::Join": [ "", [
                    "/",
                    { "Ref": "ProjectStack" },
                    "/",
                    { "Ref": "DeploymentName" },
                    "/"
                ]] },
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Action": "sts:AssumeRoleWithWebIdentity",
                            "Principal": {
                                "Federated": "cognito-identity.amazonaws.com"
                            }
                        }
                    ]
                }
            }
        },
        "PlayerAccess": {
            "Type": "Custom::PlayerAccess",
            "Properties": {
                "ServiceToken": { "Ref": "ProjectResourceHandler" },
                "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "DeploymentStack": { "Ref": "DeploymentStackArn" }
            },
            "DependsOn": [ "Player" ]
        },
        "PlayerLoginRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Action": "sts:AssumeRoleWithWebIdentity",
                            "Principal": {
                                "Federated": "cognito-identity.amazonaws.com"
                            }
                        }
                    ]
                },
                "Policies": [
                    {
                        "PolicyName": "ExchangeTokenAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Sid": "PlayerLoginExecution",
                                    "Effect": "Allow",
                                    "Action": [ "lambda:InvokeFunction" ],
                                    "Resource": { "Ref": "ProjectPlayerAccess
TokenExchangeHandler" }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "PlayerLoginIdentityPool": {
            "Type": "Custom::CognitoIdentityPool",
            "Properties": {
                "ServiceToken": { "Ref": "ProjectResourceHandler" },
                "AllowUnauthenticatedIdentities": "true",
                "UseAuthSettingsObject": "false",
                "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "IdentityPoolName": "PlayerLogin",
                "Roles": {
                    "unauthenticated": { "Fn::GetAtt": [ "PlayerLoginRole",
"Arn" ] }
                }
            }
        },
        "PlayerAccessIdentityPool": {
            "Type": "Custom::CognitoIdentityPool",
            "Properties": {
                "ServiceToken": { "Ref": "ProjectResourceHandler" },
                "AllowUnauthenticatedIdentities": "true",
                "UseAuthSettingsObject": "true",
                "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "IdentityPoolName": "PlayerAccess",
                "Roles": {
                    "unauthenticated":  { "Fn::GetAtt":  [ "Player", "Arn"] },

                    "authenticated": { "Fn::GetAtt": [ "Player", "Arn" ] }
                }
            }
        }
    }
}
```

Parameters

The deployment access stack defines parameters that identify the deployment stack and other resources that are needed to setup security for the deployment. A value for each of these parameters is provided by the `project-template.json (p. 146)` file.

Resources

This section describes the resources defined in the example `deployment-access-template.json` file.

OwnerPolicy Resource

The `OwnerPolicy` resource describes a IAM Managed Policy that gives owner level access to the deployment. The AWS account administrator always has full access to the deployment, but may want to limit other users' access to specific deployments. That can be done by attaching `OwnerPolicy` to an IAM User or use the Owner role, which is also defined by the deployment access template.

Owner access includes the following:

- The ability to update the deployment stack.
- Full access to the feature resources created for the deployment.

For more information, see Project Access Control (p. 188).

Owner Resource

The `Owner` resource describes an IAM role with the `OwnerPolicy` attached.

For more information, see Project Access Control (p. 188).

Player Resource

The `Player` resource describes the IAM role that determines the access granted to the player. For example, for the game to invoke a Lambda function, the player must be allowed the `lambda:InvokeFunction` action on the Lambda function resource.

The role's policies are determined by the `PlayerAccess` metadata elements found on resources in the project's feature templates (see `feature-template.json (p. 162)`). The role's policies are updated by the `PlayerAccess` custom resources that appear in the `deployment-access-template.json (p. 154)` and in the `feature-template.json (p. 162)` files. The `PlayerAccessIdentityPool` Amazon Cognito identity pool resource allows players to assume this role.

For more information, see PlayerAccessIdentityPool Resource (p. 162) and Access Control and Player Identity (p. 187).

PlayerAccess Resource

The `PlayerAccess` resource describes a PlayerAccessIdentityPool Resource (p. 162). This resource is responsible for configuring the player role using the PlayerAccess metadata found on the resources to which the player should have access.

For more information, see Access Control and Player Identity (p. 187).

PlayerLoginRole Resource

The `PlayerLoginRole` resources describes the IAM role that is temporarily assumed by the player as part of the login process.

For more information, see Access Control and Player Identity (p. 187).

PlayerLoginIdentityPool Resource

The `PlayerLoginIdentityPool` resource describes the Amazon Cognito identity pool that provides the player with a temporary identity during the login process.

For more information, see Access Control and Player Identity (p. 187).

PlayerAccessIdentityPool Resource

The `PlayerAccessIdentityPool` resource describes the Amazon Cognito identity pool that provides the player with a temporary identity during the login process.

For more information, see Access Control and Player Identity (p. 187).

project-code

The `project-code` subdirectory contains the source code for the AWS CloudFormation Custom Resource handler that is used in the project's AWS CloudFormation templates. For information about custom resources, see Custom Resources (p. 183).

It also contains the code that implements the token exchange step of the player login process. For more information, see Access Control and Player Identity (p. 187).

feature\*{feature}* subdirectories

The AWS resources used by the game are organized into separate features, as represented by the *{feature}* subdirectory under the `feature` directory. The `feature` directory may contain any number of *{feature}* subdirectories.

feature-template.json

A `feature-template.json` file is an AWS CloudFormation template that defines the AWS resources associated with each feature. You can specify any AWS resource type supported by AWS CloudFormation in your `feature-template.json` file. For a list of the available resource types, see the AWS CloudFormation AWS Resource Types Reference.

The example `feature-template.json` file that follows defines a `SayHello` Lambda function that is executed by the game to generate a greeting for a player. The generated message is stored in a DynamoDB table. This example feature is created for you by the `lmbr_aws` initialize-project (p. 174) command.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Parameters": {
        "ProjectResourceHandler": {
            "Type": "String",
            "Description": "Service token of the custom resource handler."
        },
        "ConfigurationBucket": {
            "Type": "String",
            "Description": "Bucket that contains configuration data."
        },
        "ConfigurationKey": {
            "Type": "String",
           "Description": "Location in the configuration bucket of configuration
data."
        },
        "ReadCapacityUnits": {
            "Type": "Number",
            "Default": "1",
            "Description": "Number of game state reads per second."
        },
        "WriteCapacityUnits": {
            "Type": "Number",
            "Default": "1",
            "Description": "Number of game state writes per second."
        },
        "Greeting": {
            "Type": "String",
            "Default": "Hello",
            "Description": "Greeting used by the SayHello Lambda function."
        }
    },
    "Resources": {
        "Messages": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "PlayerId",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "PlayerId",
                        "KeyType": "HASH"
                    }
                ],
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": { "Ref": "ReadCapacityUnits" },
                    "WriteCapacityUnits": { "Ref": "WriteCapacityUnits" }
                }
            },
            "Metadata": {
                "CloudCanvas": {
                    "FunctionAccess": [
                        {
                            "FunctionName": "SayHello",
                            "Action": "dynamodb:PutItem"
                        }
                    ]
                }
            }
        },
        "SayHelloConfiguration": {
            "Type": "Custom::LambdaConfiguration",
            "Properties": {
                "ServiceToken": { "Ref": "ProjectResourceHandler" },
                "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "FunctionName": "SayHello",
                "Runtime": "python2.7",
                "Settings": {
                    "MessagesTable": { "Ref": "Messages" },
                    "Greeting": { "Ref": "Greeting" }
                }
            }
        },
        "SayHello": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Example of a function called by the game to
write data into a DynamoDB table.",
                "Handler": "main.say_hello",
                "Role": { "Fn::GetAtt": [ "SayHelloConfiguration", "Role" ] },

               "Runtime": { "Fn::GetAtt": [ "SayHelloConfiguration", "Runtime"
 ] },
                "Code": {
                    "S3Bucket": { "Fn::GetAtt": [ "SayHelloConfiguration",
"ConfigurationBucket" ] },
                  "S3Key": { "Fn::GetAtt": [ "SayHelloConfiguration", "Config
urationKey" ] }
                }
            },
            "Metadata": {
                "CloudCanvas": {
                    "PlayerAccess": {
                        "Action": "lambda:InvokeFunction"
                    }
                }
            }
        },
        "PlayerAccess": {
            "Type": "Custom::PlayerAccess",
            "Properties": {
                "ServiceToken": { "Ref": "ProjectResourceHandler" },
                "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "FeatureStack": { "Ref": "AWS::StackId" }
            },
            "DependsOn": "SayHello"
        }
    }
}
```

Feature Template Parameters

This section describes the parameters defined in the example `feature-template.json` file.

ProjectResourceHandler Parameter

The `ProjectResourceHandler` parameter identifies the custom resource handler Lambda function used for the project. The parameter value is set by Lumberyard provided tools when you use the template to update the AWS CloudFormation stack.

ConfigurationBucket Parameter

The `ConfigurationBucket` parameter identifies the configuration bucket. The parameter value is set by Lumberyard provided tools when the template is used to update the AWS CloudFormation stack.

ConfigurationKey Parameter

The `ConfigurationKey` parameter identifies the location of configuration data in the configuration bucket. The parameter value is set to a unique value by Cloud Canvas each time it uses the template to update the AWS CloudFormation stack.

ReadCapacityUnits and WriteCapacityUnits Parameters

The `ReadCapacityUnits` and `WriteCapacityUnits` parameters are used to configure the `Messages` resource defined by the template. Values for parameters such as these are typically provided by the **project-settings.json** (p. 143) and project-template.json (p. 146) file and can be customized for each deployment.

Feature Template Resources

This section describes the resources defined in the example `feature-template.json` file.

Messages Resource

The `Messages` resource describes a DynamoDB Table. See AWS::DynamoDB::Table for a description of the AWS CloudFormation DynamoDB table resource definition format.

The `Metadata.CloudCanvas.FunctionAccess` property of the resource definition is used by the `SayHelloConfiguration` custom resource to grant the `SayHello` Lambda function resource permission to write data into the table. For more information, see Lambda Function Access Control (p. 189).

SayHelloConfiguration Resource

The `SayHelloConfiguration` resource describes a LambdaConfiguration (p. 185) resource that provides various configuration inputs for the `SayHello` Lambda function resource.

The `Settings` property for this resource is used to pass configuration data to the `SayHello` Lambda function. For more information, see LambdaConfiguration (p. 185).

SayHello Resource

The `SayHello` resource describes a Lambda function resource that implements some of the game logic. See AWS::Lambda::Function for a description of the AWS CloudFormation Lambda function resource definition format.

The Lambda function's `Execution` Role, which determines the AWS permissions the function has when it executes, is created by the `SayHelloConfiguration` resource, which uses the `Metadata.CloudCanvas.FunctionAccess` properties that appear on the resources that the function can access.

The `Metadata.CloudCanvas.PlayerAccess` property of the resource definition determines the access that players have to the `SayHello` resource. In this case, they can only invoke the lambda function.

PlayerAccess Resource

The `PlayerAccess` resource in the feature template is a PlayerAccess (p. 186) custom resource. It grants players access to resources specified by the `Metadata.CloudCanvas.PlayerAccess` properties on the definitions of the resources to which they have access.

Note that the `PlayerAccess DependsOn` property lists the resources that define this metadata property. This ensures that AWS CloudFormation creates or updates the `PlayerAccess` resources after the resources with the metadata property have been created or updated.

feature-code

The `feature-code` subdirectory is present when a feature template defines Lambda function resources. This directory is where you put the source files that implement those functions.

Lumberyard provided tools uploads the code from this directory when using the template to update the AWS CloudFormation stack.

Resource Deployments

You implement deployments using AWS CloudFormation stacks. You create and manage the stacks using tools provided by Lumberyard.

A project may define any number of deployments, up to the limits imposed by AWS CloudFormation (for more information, see AWS CloudFormation Limits). Each deployment contains a completely independent set of the resources that the game requires. For example, you can have separate development, test, and release deployments so that your development and test teams can work independently of the deployment used for the released version of the game.

An AWS account that hosts a Lumberyard project contains the following resources:

- *`{project}`* – An AWS CloudFormation stack that acts as a container for all the project's deployment stacks.
- *`{project}`*`-Configuration` – An S3 bucket used to store configuration data.
- *`{project}`*`-ProjectResourceHandler` – A Lambda function that implements the handler for the custom resources used in the templates. See Custom Resources (p. 183).
- *`{project}`*`-ProjectResourceHandlerExecution` – An IAM role that grants the permissions used by the `ProjectResourceHandler` Lambda function when it is executing.
- *`{project}`*`-ProjectPlayerAccessTokenExchangeHandler` – A Lambda function that implements the token exchange step in the player login process. For more information, see Access Control and Player Identity (p. 187).
- *`{project}`*`-ProjectPlayerAccessTokenExchangeHandlerRole` – An IAM role that grants the permissions used by the `ProjectPlayerAccessTokenExchangeHandler` Lambda function when it runs.
- *`{project}`*-*`{deployment}`* – AWS CloudFormation stacks for each of the project's deployments.
- *`{project}`*-*`{deployment}`*`Access` – AWS CloudFormation stacks that control access to each of the project's deployments.

  `{project}`-*`{deployment}`*`Access-OwnerPolicy` – An IAM managed policy that grants "owner" access to a deployment. See Project Access Control (p. 188).
- *`{project}`*-*`{deployment}`*`Access-Owner` – An IAM role that grants "owner" access to a deployment. See Project Access Control (p. 188).

- *{project}*-*{deployment}*Access-Player – An IAM role that grants "player" access to a deployment. See Access Control and Player Identity (p. 187).
- *{project}*-*{deployment}*Access-PlayerLoginRole – An IAM role that grants players temporary anonymous access used during the player login process. See Access Control and Player Identity (p. 187).
- *{project}*-*{deployment}*Access-PlayerAccessIdentityPool – An Amazon Cognito identity pool used for player identity. For more information, see Access Control and Player Identity (p. 187).
- *{project}*-*{deployment}*Access-PlayerLoginIdentityPool – An Amazon Cognito identity pool that provides the temporary player identity used during the player login process. For more information, see Access Control and Player Identity (p. 187).
- *{project}*-*{deployment}*-*{feature}* – An AWS CloudFormation stack for each feature of the project.
- *{project}*-*{deployment}*-*{feature}*-*{resource}* – The resources defined by a feature. Because of how AWS CloudFormation works, parts of these names have unique identifiers appended to them. For example, for a project named MyGame with a deployment named Development and a feature named HighScore, the actual name of a Scores resource would be something like: `MyGame-Development-1FLFSUKM3MC4B-HighScore-1T7DK9P46SQF8-Scores-1A1WIH6MZKPRI`. The tools provided by Lumberyard hide these actual resource names under most circumstances.

Configuration Bucket

The configuration Amazon S3 bucket is used to store configuration data for the project. The tools provided with Cloud Canvas manage uploads to this bucket.

The configuration bucket contents are as follows.

```
/
    upload/
        {upload-id}/
            project-template.json
            project-code.zip
            deployment/
                {deployment}/
                    deployment-template.json
                    feature/
                        {feature}/
                            feature-template.json
                            feature-code.zip
                            feature-code.zip.{function-name}.configured
    player-access/
        auth-settings.json
```

All the `/upload/`*`{upload-id}`*`/*` objects in this bucket , except the `*.configured` objects, are uploaded from the *`{game}`*`/AWS` directory by the Cloud Canvas tools when stack management operations are performed. The uploads for each operation get assigned a unique *`{upload-id}`* value to prevent concurrent operations from impacting each other.

The `feature-code.zip.`*`{function-name}`*`.configured` objects in this bucket are created by the `LambdaConfiguration` custom resources when settings are injected into the code. See LambdaConfiguration (p. 185) for more information.

The `/player-access/auth-settings.json` file stores the security credentials used to implement player login by using social networks such as Facebook or by using the player's Amazon credentials. This file is created and updated by the `lmbr_aws` add-login-provider (p. 172), update-login-provider (p. 181), and remove-login-provider (p. 179) commands.

Resource Mappings

Resource mappings map the friendly names used in a game's Resource Definitions (p. 143) to the actual names of the resources created for one or more specific Resource Deployments (p. 166). For example, a DynamoDB table name like DailyGiftTable would get mapped to a name like `SamplesProject-DontDieDeployment-78AIXR0N0O4N-DontDieAWS-1I1ZC6YO7KU7F-DailyGiftTable-1G4G33K16D8ZS` where `SamplesProject` is the name of the project, `DontDieDeployment` is the name of a deployment, and `DontDieAWS` is the name of a feature. The `78AIXR0N0O4N`, `1I1ZC6YO7KU7F` and `1G4G33K16D8ZS` parts of the resource name are inserted by AWS CloudFormation to guarantee that the resource name is unique over time. Thus, even if a resource is deleted and a new one with the same logical name is created, the physical resource ID will be different.

Usually different deployments, and consequently different mappings, are used for game development and for the released version of a game. Furthermore, different development, test, and feature teams often work with their own deployments so that each team has distinct mappings.

The deployment used by default during development is specified in the *{root}*\*{game}*\AWS\project-settings.json (p. 143) file and can be overridden for each user by the *{root}\{game}*\pc\user\AWS\user-settings.json (p. 145) file. You can change these defaults using the `lmbr_aws` default-deployment (p. 173) command and by the **AWS Cloud Canvas**, **Permissions and deployments** menu item in Lumberyard Editor.

The mappings used during development when the game is launched from the Lumberyard IDE by pressing **Ctrl+G** are stored in the `user-settings.json` (p. 145) file just mentioned. This file is updated automatically when the default deployment changes, when the default deployment is updated, and when Lumberyard Editor is started. It can be refreshed manually by using the `lmbr_aws` update-mappings (p. 181) command.

When a game launcher application created in Lumberyard launches a release build of a game, the mappings for the game are stored in the *{root}\{game}*\Config\awsLogicalMappings.json file. These mappings can be updated manually using the `lmbr_aws` update-mappings (p. 181) command. You can specify the deployment for the release mappings in the `ReleaseDeployment` property of the **project-settings.json** (p. 143) file.

In both cases, the AWS Client Configuration (p. 195) system is configured with the mapping data. You can use this configuration in AWS flow nodes, with the AWS C++ SDK, and in Lambda functions.

Using Mappings in AWS Flow Nodes

AWS flow nodes that define `TableName` (DynamoDB), `FunctionName` (Lambda), `QueueName` (Amazon SQS), `TopicARN` (Amazon SNS), or `BucketName` (Amazon S3) ports work with mappings. Set the port to a value like *{feature}*.*{resource}* where *{feature}* is the name of the feature that defines the resource, and where *{resource}* is the name of the resource that appears in the *Resources* section of the feature's `feature-template.json` file. The AWS flow nodes use the AWS Client Configuration (p. 195) system to look up the actual resource name by referring to the names in the *{feature}*.*{resource}* format.

Using Mappings with the AWS C++ SDK

The AWS Client Configuration (p. 195) system supports the configuration of service specific clients using arbitrary application defined names. The Lumberyard IDE uses names in the *{feature}*.*{resource}* format to define configurations that map from those friendly names to the actual resource name that must be provided when calling AWS C++ SDK APIs. This configuration can be retrieved and used by using C++ code like the following.

Code example: mapping resource names in C++

```
#include <LmbrAWS\IAWSClientManager.h>
#include <LmbrAWS\ILmbrAWS.h>
#include <aws\lambda\LambdaClient.h>
#include <aws\lambda\model\InvokeRequest.h>
#include <aws\lambda\model\InvokeResult.h>
#include <aws\core\utils\Outcome.h>

void InvokeSayHello()
{
   LmbrAWS::IClientManager* clientManager = pEnv->pLmbrAWS->GetClientManager();


    // Use {feature}.{resource} format name to lookup the configured client.
    LmbrAWS::Lambda::FunctionClient client = clientManager->GetLambdaMan
ager().CreateFunctionClient("HelloWorld.SayHello");

   // Get the Lambda function resource name using the client.GetFunctionName()
 method.
    Aws::Lambda::Model::InvokeRequest req;
    req.SetFunctionName(client.GetFunctionName());

   // Invoke the Lambda function asynchronously (async isn't required, but you
 should never
    // do network I/O in the main game thread).
    client->InvokeAsync(req,
        []( const Aws::Lambda::LambdaClient*,
            const Aws::Lambda::Model::InvokeRequest&,
            const Aws::Lambda::Model::InvokeOutcome& outcome,
            const std::shared_ptr<const Aws::Client::AsyncCallerContext>&)
        {
            if(outcome.IsSuccess())
            {
                const Aws::Lambda::Model::InvokeResult& res = outcome.GetRes
ult();
                // TODO: process the result
            }
        }
    );

}
```

Using Mappings in Lambda Functions

Lambda function resources defined as part of a feature often need to access other resources defined by that feature. To do this, the function code needs a way to map a friendly resource name to the actual resource name used in AWS API calls. The `LambdaConfiguration` resource provides a way to such mappings, as well as other settings, to the lambda code. For more information, see LambdaConfiguration (p. 185).

Lumberyard Editor Tools

This section describes the Cloud Canvas functionality currently available from Lumberyard Editor.

Lumberyard Editor provides an **AWS** menu with the following submenus:

- **Open an AWS Console** – Find quick links to AWS consoles such as Amazon GameLift, Amazon Cognito, DynamoDB, Amazon S3, and Lambda. The links use your currently active profile.
- **Cloud Canvas** – Use the **Permissions and deployments** option to manage your AWS profiles and choose your default deployment.

Cloud Canvas Command Line

Cloud Canvas provides the *{root}*`\lmbr_aws.cmd` command line tool for working with AWS resources.The tool invokes Python code that is located in the *{root}*`\Tools\lmb_aws` directory.

Syntax

```
lmbr_aws {command} {command-arguments}
```

{command} is one of commands in the command summary section that follows. *{command-arguments}* are the arguments accepted by the command. Arguments common to most commands are listed in the Common Arguments (p. 171) section. Arguments unique to a command are listed in the detail section for the command.

Configuration

The tool gets its default AWS configuration from the same `~/.aws/credentials` and `~/.aws/config` files as the AWS command line tools (for information, see Configuring the AWS Command Line Interface). The tool does not have a dependency on the AWS command line interface installed.

Environment Variables

As with the AWS command line tools, the default AWS configuration can be overridden by using the following environment variables.

- `AWS_ACCESS_KEY_ID` The access key for your AWS account.
- `AWS_SECRET_ACCESS_KEY` The secret key for your AWS account.
- `AWS_DEFAULT_REGION` The default region to use; for example, `us-east-1`.
- `AWS_PROFILE` The default credential and configuration profile to use, if any.

Configuration Arguments

The following arguments can be used to override the AWS configuration from all other sources:

- `--profile` *{profile}* The AWS command line tool profile that is used.
- `--region` *{region}* The region targeted by the command.
- `--aws-access-key` *{access-key}* The AWS access key that is used.
- `--aws-secret-key` *{secret-key}* The AWS secret key that is used.

Command Summary

This topic discusses the following commands:

- add-deployment (p. 171) – Add a deployment to the project.

- add-feature (p. 172) – Add a feature to the project.
- add-login-provider (p. 172) – Add a player login provider to the Amazon Cognito identity pool configuration.
- add-profile (p. 173)– Add an AWS profile.
- default-deployment (p. 173)– Set or show the default deployment.
- default-profile (p. 174)– Set, clear, or show the default profile.
- delete-project-stack (p. 174) – Delete a project stack.
- initialize-project (p. 174) – Initialize Cloud Canvas resource management for a Lumberyard project.
- list-deployments (p. 175)– List all deployments in the local project.
- list-features (p. 176)– List all the features found in the local deployment template and in the selected deployment stack in AWS.
- list-mapping (p. 176)– Show the logical to physical resource name mappings.
- list-profiles (p. 177)– List the AWS profiles that have been configured.
- list-resources (p. 177)– List all of the resources associated with the project.
- remove-deployment (p. 178) – Remove a deployment from the project.
- remove-feature (p. 179) – Remove a feature from the project.
- remove-login-provider (p. 179) – Remove a player login provider from the Amazon Cognito identity pool configuration.
- remove-profile (p. 179)– Remove an AWS profile.
- rename-profile (p. 179)– Rename an AWS profile.
- update-deployment (p. 180) – Update a deployment.
- update-feature (p. 180) – Update a feature for the specified deployment.
- update-login-provider (p. 181) – Update a player login provider in the Amazon Cognito identity pool configuration.
- update-mappings (p. 181) – Update the mappings to reflect the current default deployment.
- update-profile (p. 182)– Update an AWS profile.
- update-project (p. 182) – Update all deployments of a project.

Common Arguments

Most of the `lmbr_aws` commands accept the following arguments, in addition to their own individual arguments:

- `--root-directory` *{root}* Identifies the Lumberyard install directory. The default is the current working directory.
- `--aws-directory` *{aws}* Identifies the *{game}*`\AWS` directory to use. The default is the value of the `sys_game_folder` property from *{root}*`\system.cfg` with `AWS` appended.
- `--user-directory` *{user}* Location of the user cache directory. The default is *{root}*`\Cache\`*{game}*`\AWS` where *{game}* is determined by the `sys_game_folder` setting in the *{root}*`\bootstrap.cfg` file.

Commands

Following are details of the `lmbr_aws` commands.

add-deployment

Add `DeploymentConfiguration` and AWS CloudFormation stack resources to your `project-template.json` file. The added resources will be similar to the DevelopmentConfiguration

Resource (p. 152) , Development Resource (p. 152) , and DevelopmentAccess Resource (p. 152) in the example project-template.json (p. 146) file.

In addition to the Common Arguments (p. 171), the `add-deployment` subcommand accepts the following argument:

- `--deployment-name` *`{deployment-name}`* Required. The name of the deployment to add.

add-feature

Add a `FeatureConfiguration` and AWS CloudFormation stack resources to your `deployment-template.json` file. The added resources will be similar to the HelloWorldConfiguration Resource (p. 154) and HelloWorld Resource (p. 154) in the example deployment-template.json (p. 152) file.

The command also creates a *`{game}`*\feature\*`{feature-name}`* directory with a default `feature-template.json` file and `feature-code` subdirectory.

In addition to the Common Arguments (p. 171), the `add-feature` subcommand accepts the following argument:

- `--feature-name` *`{feature-name}`*

  Required. The name of the feature to add.

add-login-provider

Add a player login provider to the Amazon Cognito identity pool configuration. Login providers allow your game's players to log in using their social network identity, such as Facebook or Twitter, or using their Amazon user identity. For more information, see Access Control and Player Identity (p. 187).

In addition to the Common Arguments (p. 171), the `add-login-provider` subcommand accepts the following arguments:

- `--provider-name` *`{provider-name}`*

  Required. The name of the provider. The name must be `amazon`, `google` or `facebook`, or, if you are using a generic OpenID provider, a name that you choose.
- `--app-id` *`{application-id}`*

  Required. The application id from your login provider (this is usually different from your client ID).
- `--client-id` *`{client-id}`*

  Required. The unique application client ID for the login provider.
- `--client-secret` *`{client-secret}`*

  Required. The secret key to use with your login provider.
- `--redirect-uri` *`{redirect-uri}`*

  Required. The redirect URI to use with your login provider.
- `--provider-uri` *`{provider-uri}`*

  Optional. The URI for a generic open ID connect provider. This is only use for generic OpenID providers.
- `--provider-port` *`{provider-port}`*

  Optional. The port your provider listens on for its API. This is only used for generic OpenID providers.
- `--provider-path` *`{provider-path}`*

Optional. The path portion of your provider's URI. This is only used for generic OpenID providers.

This command saves its configuration in a `player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

Note
You must run `lmbr_aws update-project` after running this command so that the PlayerAccessIdentityPool Resource (p. 162) configuration is updated to reflect the change.

add-profile

Add an AWS profile.

In addition to the Common Arguments (p. 171), the `add-profile` subcommand accepts the following arguments:

- `--aws-access-key`*`{accesskey}`*

  Required. The AWS access key associated with the added profile.
- `--aws-secret-key`*`{secretkey}`*

  Required. The AWS secret key associated with the added profile.
- `--profile` *`{profilename}`*

  Required. The name of the AWS profile to add.
- `--make-default`

  Optional. Make the new profile the default profile.

default-deployment

Set or show the default deployment.

In addition to the Common Arguments (p. 171), the `default-deployment` subcommand accepts the following arguments:

- `--set` *`{deployment}`*

  Optional. Sets the default to the provided deployment name.
- `--clear`

  Optional. Clears the defaults.
- `--show`

  Optional. Shows the defaults.
- `--project`

  Optional. Applies `--set` and `--clear` to the project default instead of the user default. Ignored for `--show`.

  Only one of the `--set`, `--clear`, and `--show` arguments is allowed.

  If `--set` or `--clear` is specified, this command updates the *`{root}`*`\user\AWS\user-settings.json` file. If `--project` is provided, the *`{root}`*`\`*`{game}`*`\AWS\project-settings.json` file is updated.

default-profile

Set, clear, or show the default profile.

In addition to the Common Arguments (p. 171), the `default-profile` subcommand accepts the following arguments:

- `--set` *`{deploymentname}`*

  Optional. Set the default profile to the provided deployment name.
- `--clear`

  Optional. Clear the default profile.
- `--show`

  Optional. Show the default profile.

delete-project-stack

Run an AWS CloudFormation delete stack operation on the stack specified. The command deletes a project stack, including all its child deployment stacks, child feature stacks, and the resources defined by those stacks.

In addition to the Common Arguments (p. 171), the `delete-project-stack` subcommand accepts the following arguments:

- `--project-stack-id` *`{stack-id}`*

  Required. The ARN ID of the project stack to be deleted. This argument is required. The argument must not be the same as the value of the `StackId` property in the *`{root}`*`\`*`{game}`*`\AWS\project-settings.json` file.
- `--confirm-resource-deletion`

  Optional. Confirms with the user that this command will delete all the AWS resources defined by the project, deployment, and feature templates. Also disables the prompt for confirmation done by the command during the command's execution.

  AWS CloudFormation cannot delete stacks that define Amazon S3 buckets that contain data. To allow project stacks to be deleted, the `project-template.json` file specifies a `DeletionPolicy` of Retain for the configuration bucket. This causes AWS CloudFormation to not delete the bucket when the project stack is deleted. After the project stack has been deleted, the command removes all the objects from the configuration bucket and then deletes the bucket.

initialize-project

Initialize Cloud Canvas resource management for a Lumberyard project. This includes creating a set of default Resource Definitions (p. 143) in the *`{root}`*`\`*`{game}`*`\AWS` directory and creating the project's initial deployment.

In addition to the Common Arguments (p. 171), the `initialize-project` subcommand accepts the following arguments:

- `--stack-name` *`{stack-name}`*

  Optional. The name used for the project's AWS CloudFormation stack. The default is the name of the *`{game}`* directory.

- `--confirm-aws-usage`

  Optional. Confirms that you know this command will create AWS resources for which you may be charged and that it may perform actions that can affect permissions in your AWS account. Also disables the prompt for confirmation during the command's execution.
- `--enable-capability {capability} [{capability} ...]`

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that can affect permissions in your AWS account. For those stacks, you must explicitly acknowledge their capabilities by specifying this parameter. Possible values include `CAPABILITY_IAM`.
- `--files-only`

  Optional. Writes the default configuration data to the *{game}*`\AWS` directory and exits. The directory must be empty or must not exist.

How `initialize-project` works

1. The `initialize-project` command creates the project's AWS CloudFormation stack using a bootstrap template that defines only the Configuration Bucket (p. 167) resource.
2. The project-template.json (p. 146), deployment-template.json (p. 152), and feature-template.json (p. 162) files and the zipped up contents of the project-code (p. 162) and feature-code (p. 166) are uploaded to the Configuration Bucket (p. 167).
3. An AWS CloudFormation stack update operation is performed by using the uploaded `project-template.json` file, which in turn updates the deployment stacks using the uploaded `deployment-template.json` and `feature-template.json` files. The `project-code` and `feature-code .zip` files are used when creating the Lambda function resources defined by the templates.

Note

- If the *{root}*`\`*{game}*`\AWS` directory is empty or does not exist, `initialize-project` creates the directory if necessary and copies the contents of the *{root}*`\Tools\lmb_aws\lmbr_aws\default-content` directory to that directory.
- `initialize-project` fails if a stack with the specified name already exists in the configured AWS account and region.
- `initialize-project` fails if the *{root}*`\`*{game}*`\AWS\project-settings.json` file has a non-empty `StackId` property. The `StackId` property will be set to the project's AWS CloudFormation stack ID.

list-deployments

List all deployments in the local project.

Example output:

```
Name                 Status            Reason
                                       Timestamp           Id
-----------------  ---------------   ------------------------------------------
---------------------------------  -----------------  -----------------------
-----------------------------------------------------------------------------
---------------------
AnotherDeployment  CREATE_PENDING    Resource is defined in the local project
```

```
template but does not exist in AWS.
Development        CREATE_COMPLETE
                                 03/04/16 18:43:11  arn:aws:cloudformation:us-
east-1:<ACCOUNTID>:stack/foo-hw-Development-ZDLXUB7FKR94/8e6492f0-e248-11e5-
8e7e-50d5ca6e60ae


User Default Deployment:    (none)
Project Default Deployment: Development
Release Deployment:         (none)
```

list-features

List all the features found in the local deployment template and in the selected deployment stack in AWS.

In addition to the Common Arguments (p. 171), the `list-features` subcommand accepts the following argument:

- `--deployment-name` *`{deployment-name}`*

  Optional. The name of the deployment to list features for. If not given, the default deployment is used.

Example output:

```
Name            Status           Reason
                                    Timestamp          Id
--------------  ---------------  ------------------------------------------------
----------------------------------  -----------------  ------------------------
---------------------------------------------------------------------------------
---------------------------------------------
AnotherFeature  CREATE_PENDING   Resource is defined in the local deployment
template but does not exist in AWS.
HelloWorld      CREATE_COMPLETE
                                 03/04/16 18:42:57  arn:aws:cloudformation:us-
east-1:<ACCOUNTID>:stack/foo-hw-Development-ZDLXUB7FKR94-HelloWorld-WSGZ15EU
WX52/9b909d20-e238-11e5-a98d-50fae987c09a
```

list-mapping

Show the logical to physical resource name mappings.

Example output:

```
Name                                       Type                         Id
----------------------------------------  ---------------------------  --------
--------------------------------------------------------
HelloWorld.SayHello                        AWS::Lambda::Function        foo-hw-
Development-ZDLXUB7FKR94-HelloWo-SayHello-1FADMFNE5M1CO
PlayerAccessIdentityPool                  Custom::CognitoIdentityPool  us-east-
1:108f6d6a-f929-4212-9947-a03269b9582e
PlayerLoginIdentityPool                   Custom::CognitoIdentityPool  us-east-
1:3020e175-0ddd-4860-8dad-1db57162cbb2
ProjectPlayerAccessTokenExchangeHandler  AWS::Lambda::Function        foo-hw-
ProjectPlayerAccessTokenExchangeHandler-1BG6JJ94IZAUV
account_id                               Configuration                 <ACCOUNTID>
```

```
region                                    Configuration               us-east-
1
```

list-profiles

List the AWS profiles that have been configured.

list-resources

List all of the resources associated with the project.

In addition to the Common Arguments (p. 171), the `list-resources` subcommand accepts the following arguments:

- `--stack-id` *{stackid}*

  Optional. The ARN of the stack to list resources for. Defaults to project, deployment, or feature stack id as determined by the `--deployment-name` and `--feature-name` parameters.
- `--deployment-name` *{deploymentname}*

  Optional. The name of the deployment to list resources for. If not specified, lists all the project's resources.
- `--feature-name` *{feature-name}*

  Optional. The name of the feature to list resources for. If specified, `deployment-name` must also be specified. If not specified, all deployment or project resources are listed.

Example output:

```
Name                                          Type
 Status           Timestamp          Id
--------------------------------------------  -------------------------------
 ---------------  -----------------  -----------------------------------------
------------------------------------------------------------------------------
---------------------------
Configuration                                 AWS::S3::Bucket
 CREATE_COMPLETE  03/04/16 18:38:25  foo-hw-configuration-vxaq1g44s0ef
Development                                   AWS::CloudFormation::Stack
 CREATE_COMPLETE  03/04/16 18:43:11  arn:aws:cloudformation:us-east-1:<ACCOUNT
ID>:stack/foo-hw-Development-ZDLXUB7FKR94/8e6492f0-e238-11e5-8e7e-50d5ca6e60ae
Development.HelloWorld                        AWS::CloudFormation::Stack
 CREATE_COMPLETE  03/04/16 18:42:57  arn:aws:cloudformation:us-east-1:<ACCOUNT
ID>:stack/foo-hw-Development-ZDLXUB7FKR94-HelloWorld-WSGZ15EUWX52/9b909d20-e238-
11e5-a98d-50fae987c09a
Development.HelloWorld.Messages               AWS::DynamoDB::Table
 CREATE_COMPLETE  03/04/16 18:41:24  foo-hw-Development-ZDLXUB7FKR94-HelloWorld-
WSGZ15EUWX52-Messages-W8398CX6EB7C
Development.HelloWorld.PlayerAccess           Custom::PlayerAccess
 CREATE_COMPLETE  03/04/16 18:42:54  CloudCanvas:PlayerAccess:foo-hw-Development-
ZDLXUB7FKR94-HelloWorld-WSGZ15EUWX52
Development.HelloWorld.SayHello               AWS::Lambda::Function
 CREATE_COMPLETE  03/04/16 18:42:45  foo-hw-Development-ZDLXUB7FKR94-HelloWo-
SayHello-1FADMFNE5M1CO
Development.HelloWorld.SayHelloConfiguration  Custom::LambdaConfiguration
 CREATE_COMPLETE  03/04/16 18:42:39  CloudCanvas:LambdaConfiguration:foo-hw-
Development-ZDLXUB7FKR94-HelloWorld-WSGZ15EUWX52:SayHello:6e3be3f1-933b-47b7-
```

```
b3f6-21a045cbdda7
Development.HelloWorldConfiguration          Custom::FeatureConfiguration
 CREATE_COMPLETE  03/04/16 18:40:39  CloudCanvas:LambdaConfiguration:foo-hw-
Development-ZDLXUB7FKR94:HelloWorld
DevelopmentAccess                            AWS::CloudFormation::Stack
 CREATE_COMPLETE  03/04/16 18:44:58  arn:aws:cloudformation:us-east-1:<ACCOUNT
ID>:stack/foo-hw-DevelopmentAccess-14RNG9550IZMJ/f56ff7f0-e238-11e5-a77e-
50d5cd148236
DevelopmentAccess.Owner                      AWS::IAM::Role
 CREATE_COMPLETE  03/04/16 18:44:38  foo-hw-DevelopmentAccess-14RNG9550IZMJ-
Owner-1H1MHLAZOKELJ
DevelopmentAccess.OwnerPolicy                AWS::IAM::ManagedPolicy
 CREATE_COMPLETE  03/04/16 18:43:23  arn:aws:iam::<ACCOUNTID>:policy/foo-
hw/Development/foo-hw-DevelopmentAccess-14RNG9550IZMJ-OwnerPolicy-1CE1PRKWZCVRW
DevelopmentAccess.Player                     AWS::IAM::Role
 CREATE_COMPLETE  03/04/16 18:44:33  foo-hw-DevelopmentAccess-14RNG9550IZMJ-
Player-1JXYH5PPO434S
DevelopmentAccess.PlayerAccess               Custom::PlayerAccess
 CREATE_COMPLETE  03/04/16 18:44:49  CloudCanvas:PlayerAccess:foo-hw-Develop
mentAccess-14RNG9550IZMJ
DevelopmentAccess.PlayerAccessIdentityPool   Custom::CognitoIdentityPool
 CREATE_COMPLETE  03/04/16 18:44:41  us-east-1:108f6d6a-f928-4212-9947-
a03269b9582e
DevelopmentAccess.PlayerLoginIdentityPool    Custom::CognitoIdentityPool
 CREATE_COMPLETE  03/04/16 18:44:43  us-east-1:3020e175-0ded-4860-8dad-
1db57162cbb2
DevelopmentAccess.PlayerLoginRole            AWS::IAM::Role
 CREATE_COMPLETE  03/04/16 18:44:33  foo-hw-DevelopmentAccess-14RNG95-PlayerLo
ginRole-70M854BKMJBL
DevelopmentConfiguration                     Custom::DeploymentConfiguration
 CREATE_COMPLETE  03/04/16 18:40:17  CloudCanvas:DeploymentConfiguration:foo-
hw:Development
ProjectPlayerAccessTokenExchangeHandler      AWS::Lambda::Function
 CREATE_COMPLETE  03/04/16 18:40:39  foo-hw-ProjectPlayerAccessTokenExchange
Handler-1BG6JJ84IZAUV
ProjectPlayerAccessTokenExchangeHandlerRole  AWS::IAM::Role
 CREATE_COMPLETE  03/04/16 18:40:33  foo-hw-ProjectPlayerAccessTokenExchange
HandlerRo-T0E7MYI0B67N
ProjectResourceHandler                       AWS::Lambda::Function
 CREATE_COMPLETE  03/04/16 18:40:08  foo-hw-ProjectResourceHandler-XAP5CBAMQCYP
ProjectResourceHandlerExecution              AWS::IAM::Role
 CREATE_COMPLETE  03/04/16 18:40:02  foo-hw-ProjectResourceHandlerExecution-
K24FL427PVZM
```

remove-deployment

Remove a deployment's `DeploymentConfiguration` and AWS CloudFormation stack resources from your `project-template.json` file.

In addition to the Common Arguments (p. 171), the `remove-deployment` subcommand accepts the following argument:

- `--deployment-name` *{deployment-name}*

  Required. The name of the deployment to remove.

remove-feature

Remove a feature's `FeatureConfiguration` and AWS CloudFormation stack resources from your `deployment-template.json` file.

The command does not delete the *`{game}`*\feature\*`{feature-name}`* directory.

In addition to the Common Arguments (p. 171), the `remove-feature` subcommand accepts the following argument:

- `--feature-name` *`{feature-name}`*

  Required. The name of the feature to be removed.

remove-login-provider

Remove a player login provider from the Amazon Cognito identity pool configuration.

In addition to the Common Arguments (p. 171), the `remove-login-provider` subcommand accepts the following argument:

- `--provider-name` *`{provider-name}`*

  Required. The name of the provider.

  The `remove-login-provider` command saves the configuration in a `/player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

  Note
  You must run `lmbr_aws update-project` after running this command so that the PlayerAccessIdentityPool Resource (p. 162) configuration is updated to reflect the change.

remove-profile

Remove an AWS profile.

In addition to the Common Arguments (p. 171), the `remove-profile` subcommand accepts the following argument:

- `--profile` *`{profile-name}`*

  Required. The name of the AWS profile to remove.

rename-profile

Rename an AWS profile.

In addition to the Common Arguments (p. 171), the `rename-profile` subcommand accepts the following arguments:

- `--old-name` *`{old-profile-name}`*

  Required. The name of the AWS profile to change.
- `--new-name` *`{new-profile-name}`*

Required. The new name of the AWS profile.

update-deployment

Update a deployment's AWS CloudFormation stack, including all of its child feature stacks.

In addition to the Common Arguments (p. 171), the `update-deployment` subcommand accepts the following arguments:

- `--deployment-name`

  Required. The name of the deployment to update.
- `--confirm-aws-usage`

  Optional. Confirms that you know this command will create AWS resources for which you may be charged and that it may perform actions that can affect permissions in your AWS account. It also disables the default confirmation prompt that occurs during the command's execution.
- `--enable-capability` *`{capability}`* [*`{capability}`* ...]

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that can affect permissions in your AWS account. For these stacks, you must explicitly acknowledge their capabilities by specifying this parameter. Possible values include `CAPABILITY_IAM`.

  The deployment-template.json (p. 152) and feature-template.json (p. 162) files and the zipped up contents of the feature-code (p. 166) are uploaded to the Configuration Bucket (p. 167). An AWS CloudFormation stack update operation is then performed by using the uploaded `deployment-template.json` file, which in turn updates the feature stacks using the uploaded `feature-template.json` files. The `feature-code .zip` files are used when creating the Lambda function resources defined by the templates.

update-feature

Update a feature's AWS CloudFormation stack.

In addition to the Common Arguments (p. 171), the `update-feature` subcommand accepts the following arguments:

- `--deployment-name`

  Required. The name of the deployment to update.
- `--feature-name`

  Required. The name of the feature to update.
- `--confirm-aws-usage`

  Optional. Confirms that you know this command will create AWS resources for which you may be charged and that it may perform actions that can affect permissions in your AWS account. It also disables the default confirmation prompt that occurs during the command's execution.
- `--enable-capability` *`{capability}`* [*`{capability}`* ...]

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that can affect permissions in your AWS account. For those stacks, you must explicitly acknowledge their capabilities by specifying this parameter. Possible values include `CAPABILITY_IAM`.

The feature-template.json (p. 162) file and the zipped up contents of the feature-code (p. 166) are uploaded to the Configuration Bucket (p. 167). An AWS CloudFormation stack update operation is then performed by using the uploaded template file. The `feature-code .zip` file is used when creating the Lambda function resources defined by the feature template.

update-login-provider

Update a player login provider in the Amazon Cognito identity pool configuration. Login providers allow your game's players to log in using their social network identity, such as Facebook or Google, or using their Amazon user identity. For more information, see Access Control and Player Identity (p. 187).

In addition to the Common Arguments (p. 171), the `update-login-provider` subcommand accepts the following arguments:

- `--provider-name` *`{provider-name}`*

  Required. The name of the updated provider. The name must be `amazon`, `google` or `facebook`, or, if you are using a generic OpenID provider, the name that you chose when the provider was added.
- `--app-id` *`{application-id}`*

  Optional. The application ID from your login provider (this is usually different from your client ID).
- `--client-id` *`{client-id}`*

  Optional. The unique application client ID for the login provider.
- `--client-secret` *`{client-secret}`*

  Optional. The secret key to use with your login provider.
- `--redirect-uri` *`{redirect-uri}`*

  Optional. The redirect URI to use with your login provider.
- `--provider-uri` *`{provider-uri}`*

  Optional. The URI for a generic open id connect provider. This argument is used only for generic OpenID providers.
- `--provider-port` *`{provider-port}`*

  Optional. The port the provider listens on for the provider's API. This argument is used only for generic OpenID providers.
- `--provider-path` *`{provider-path}`*

  Optional. The path portion of the provider's URI. This argument is used only for generic OpenID providers.

  The `update-login-provider` command saves its configuration in a `/player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

  Note
  You must run `lmbr_aws update-project` after running this command so that the PlayerAccessIdentityPool Resource (p. 162) configuration is updated to reflect the change.

update-mappings

Update the friendly name to physical resource ID mappings to reflect the current default deployment or the release deployment.

In addition to the Common Arguments (p. 171), the `update-mappings` subcommand accepts the following argument:

- `--release`

  Optional. Causes the release mappings to be updated. By default, only the mappings used when launching the game from inside the editor are updated.

  The command looks in the feature-template.json (p. 162) file for `Metadata.CloudCanvas.PlayerAccess` properties on resource definitions. It then queries AWS CloudFormation for the physical names of those resources in the current default deployment. If the `--release` option is specified, the release deployment is queried.

update-profile

Update an AWS profile.

In addition to the Common Arguments (p. 171), the `update-profile` subcommand accepts the following arguments:

- `--aws-access-key`*`{accesskey}`*

  Optional. The AWS access key associated with the updated profile. The default is to not change the AWS access key associated with the profile.
- `--aws-secret-key`*`{secretkey}`*

  Optional. The AWS secret key associated with the updated profile. The default is to not change the AWS secret key associated with the profile.
- `--profile` *`{profilename}`*

  Required. The name of the AWS profile to update.

  Note
  To make an existing profile the default profile, use the default-profile (p. 174) command.

update-project

Update the project's AWS CloudFormation stack, including all of its child deployment stacks and their child feature stacks.

In addition to the Common Arguments (p. 171), the `update-project` subcommand accepts the following arguments:

- `--confirm-aws-usage`

  Optional. Confirms that you know this command will create AWS resources for which you may be charged and that it may perform actions that can affect permission in your AWS account. Also disables the prompt for confirmation done during the command's execution.
- `--enable-capability` *`{capability}`* [*`{capability}`* ...]

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that can affect permissions in your AWS account. For those stacks, you must explicitly acknowledge their capabilities by specifying this parameter. Possible values include `CAPABILITY_IAM`.

How `update-project` works

1. The project-template.json (p. 146), deployment-template.json (p. 152), and feature-template.json (p. 162) files and the zipped up contents of the project-code (p. 162) and feature-code (p. 166) directories are uploaded to the Configuration Bucket (p. 167).
2. An AWS CloudFormation stack update operation is performed by using the uploaded `project-template.json` file, which in turn updates the deployment stacks using the uploaded `deployment-template.json` and `feature-template.json` files. The `project-code` and `feature-code .zip` files are used when creating the Lambda function resources defined by the templates.

> **Note**
> The `update-project` command fails if the *{root}*`\`*{game}*`\AWS\project-settings.json` file does not exist or does not have a valid `StackId` property.

Custom Resources

Cloud Canvas provides a number of AWS CloudFormation custom resources that can be used in the project, deployment, and feature AWS CloudFormation template files. These custom resources are implemented by the Lambda function code found in the *{root}*`\`*{game}*`\AWS\project-code` directory and the `ProjectResourceHandler` resource defined in the *{root}*`\`*{game}*`\AWS\project-template.json` file. Rather than static entities, these resources act more like library functions Each custom resource has input and output properties.

A summary list of custom resources follows.

- CognitoIdentityPool (p. 183) – Manages Amazon Cognito identity pool resources.
- DeploymentConfiguration (p. 184) – Provides configuration data for a deployment's AWS CloudFormation stack resource.
- FeatureConfiguration (p. 185) – Provides configuration data for feature's AWS CloudFormation stack resource.
- LambdaConfiguration (p. 185) – Provides configuration data for Lambda function resources and maintains the Lambda function's execution role.
- PlayerAccess (p. 186) – Maintains the policies on the player role.

CognitoIdentityPool

The `Custom::CognitoIdentityPool` resource is used in the `deployment-access-template.json` file to create and configure Amazon Cognitoidentity pool resources.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.
- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where project configuration data is located in the configuration bucket. This property causes the custom resource handler to be executed by AWS CloudFormation for every operation.
- `IdentityPoolName`

Required. The name of the identity pool.

- `UseAuthSettingsObject`

  Required. Must be either `true` or `false`. Determines whether the Amazon Cognito identity pool is configured to use the authentication providers created using the `add-login-provider` command.
- `AllowUnauthenticatedIdentities`

  Required. Must be either `true` or `false`. Determines whether the Amazon Cognito identity pool is configured to allow unauthenticated identities. See Identity Pools for more information on Amazon Cognito's support for authenticated and unauthenticated identities.
- `Roles`

  Optional. Determines the IAM role assumed by authenticated and unauthenticated users. See SetIdentityPoolRoles for a description of this property.

Output Properties

- `IdentityPoolName`

  The name of the identity pool (same as the `IdentityPoolName` input property).
- `IdentityPoolId`

  The physical resource name of the identity pool.

DeploymentConfiguration

The `Custom::DeploymentConfiguration` resource is used in the `project-template.json` to identify the location of the copy of the `deployment-template.json` file in the configuration bucket that should be used for a specific deployment.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.
- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where configuration data for the deployment is located in the configuration bucket.
- `DeploymentName`

  Required. The name of the deployment that is to be configured.

Output Properties

- `ConfigurationBucket`

  The name of the Amazon S3 bucket that contains the configuration data. This is always the same as the `ConfigurationBucket` input property.
- `ConfigurationKey`

The Amazon S3 object key prefix where the specified deployment's configuration data is located in the configuration bucket. The output `ConfigurationKey` is the input `ConfigurationKey` with the word "deployment" and the value of `DeploymentName` appended.

- `AccessTemplateURL`

  The Amazon S3 URL of the deployment's copy of the `deployment-access-template.json` in the configuration bucket. This value should be used as the deployment access stack resources's `TemplateURL` property value.
- `DeploymentTemplateURL`

  The Amazon S3 URL of the deployment's copy of the `deployment-template.json` in the configuration bucket. This value should be used as the deployment stack resources's `TemplateURL` property value.

FeatureConfiguration

The `Custom::FeatureConfiguration` resource is used in the `deployment-template.json` to identify the location of the copy of the `feature-template.json` file in the configuration bucket that should be used for a specific feature.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.
- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where the deployment configuration data is located in the configuration bucket.
- `FeatureName`

  Required. The name of the feature that is to be configured.

Output Properties

- `ConfigurationBucket`

  The name of the Amazon S3 bucket that contains the configuration data. This is always the same as the `ConfigurationBucket` input property.
- `ConfigurationKey`

  The Amazon S3 object key prefix where the specified feature's configuration data is located in the configuration bucket. This is the input `ConfigurationKey` with "feature" and FeatureName appended.
- `TemplateURL`

  The Amazon S3 URL of the feature's copy of the feature-template.json in the configuration bucket. This value should be used as the feature stack resources's `TemplateURL` property value.

LambdaConfiguration

The `Custom::LambdaConfiguration` resource is used in `feature-template.json` files to provide configuration data for Lambda function resources.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.
- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where configuration data for the feature is located in the configuration bucket.
- `FunctionName`

  Required. The friendly name of the Lambda Function resource being configured.
- `Settings`

  Optional. Values that are made available to the Lambda function code.
- `Runtime`

  Required. Identifies the runtime used for the Lambda function. Cloud Canvas currently supports the following Lambda runtimes: `nodejs`, `python2.7`.

Output Properties

- `ConfigurationBucket`

  The name of the Amazon S3 bucket that contains the configuration data. This is always the same as the `ConfigurationBucket` input property.
- `ConfigurationKey`

  The Amazon S3 object key prefix where the specified function's zipped up code is located in the configuration bucket.
- `Runtime`

  The Lambda runtime used by the function. This is always the same as the input `Runtime` property value.
- `Role`

  The ID of the Lambda function execution created for this function.

For information on how the `LambdaConfiguration` custom resource is used to allow Lambda functions to perform specified actions on specific project resources, see Lambda Function Access Control (p. 189).

PlayerAccess

The `Custom::PlayerAccess` resource is used in `feature-template.json` files to update the player role so that players have the desired access to the feature's resources. It is also used in the `deployment-access-template.json` file to update the player role so that players have the desired access to the deployment's resources.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.
- `ConfigurationKey`

Required. The Amazon S3 object key prefix where configuration data for the deployment is located in the configuration bucket. The value of this property isn't actually used, however since the Cloud Canvas tools insure that the key is different for each AWS CloudFormation operation, the presences of this property has the effect of forcing the custom resource handler to be executed by AWS CloudFormation on for every operation.

- `FeatureStack`

  Optional. The ID of the feature stack for which the player role is updated.
- `DeploymentStack`

  Optional. The ID of the deployment stack for which the pcolayer role is updated.

  Only one of the `FeatureStack` and `DeploymentStack` properties must be provided.

Output Properties

The `PlayerAccess` custom resource does not produce any output values.

PlayerAccess Metadata Format

This custom resource looks for `Metadata.CloudCanvas.PlayerAccess` properties on the project's feature resource definitions and constructs a policy which is attached to the player role. The policy allows the indicated actions on those resources. The `Metadata.CloudCanvas.PlayerAccess` property has the following form:

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "...": {
            "Type": "...",
            "Properties": {
                ...
            },
            "Metadata": {
                "CloudCanvas": {
                    "PlayerAccess": {
                      "Action": [ "{allowed-action-1}", ..., "{allowed-action-
n}" ]
                    }
                }
            }
        },
      ...
    }
}
```

The required `Action` property is the same as defined for an IAM policy and is described in detail in the IAM Policy Elements Reference. Note that a single value can be provided instead of a list of values.

Access Control and Player Identity

Cloud Canvas helps you control access to your game's AWS resources in three ways:

- Project Access Control (p. 188)

Project Access Control

It is often necessary to limit project team member access to the project's resources. This can help prevent different development teams from accidentally updating the resources being used by another development team. It is also necessary to prevent project team members from accessing the resources used by the released version of the game, both to prevent accidental changes that could impact the operation of the game but in some cases to also prevent project team members from accessing player's personal information, such as e-mail addresses, which may be stored in those resources.

The default deployment-access-template.json (p. 154) file provided by Cloud Canvas defines an OwnerPolicy Resource (p. 161) IAM managed policy resource, which allows a deployment AWS CloudFormation stack to be updated, including creating, updating, and deleting the resources defined by the project's features. This template also defines an Owner Resource (p. 161). IAM role resource that has the `OwnerPolicy` attached.

If desired, the `OwnerPolicy` resource definition in the `deployment-access-template.json` file can be modified or additional policies can be created. However, be sure that you really understand how IAM permissions work before doing so. Incorrectly using this features can make your AWS account vulnerable to attack and abuse, which could result in unexpected AWS charges (in addition to any other repercussions).

Authorize AWS Use in Lumberyard Editor

To authorize a group of developers to use AWS via Lumberyard Editor, perform the following steps.

To authorize AWS use in Lumberyard Editor

1. Create an IAM user for each developer.
2. Generate the access key and secret keys for each user.
3. Attach a policy to the IAM user that determines what that user is allowed to do. These policies are generated when a project is initialized, or you can apply your own.
4. Deliver the access key and secret key to the developer by a secure method.

   Caution
   You should not deliver access or secret keys by using email, or check them into source control. Such actions present a significant security risk.
5. In Lumberyard Editor, have each developer navigate to **AWS**, **Cloud Canvas**, **Permissions and deployments**.
6. Have the developer add a new profile that uses the access key and secret key that he or she has been provided.

Player Access Control

In order for the game to access AWS resources at runtime, it must use credentials that grant the necessary access when calling AWS APIs. This could be done by creating an IAM user with limited privileges and embedding that user's credentials (both the AWS access key and the secret key) in the game code. But AWS Amazon Cognito identity pools provide a more powerful and secure solution for this problem.

How Cloud Canvas uses Amazon Cognito identity pools is described in the Player Identity (p. 191) section.

Ultimately player access is controlled by the player role defined in the default Cloud Canvas deployment-access-template.json (p. 154) file. The policies attached to this role are set by the PlayerAccess Resource (p. 166) custom resources that appear in the feature-template.json (p. 162) files.

Lambda Function Access Control

When an AWS Lambda function is executed, it assumes an IAM role that determines the access the function has to other AWS resources. Creating and configuring such roles requires IAM permissions that cannot safely be granted to all the project's team members; doing so would allow them to circumvent the security measures that limit their access to specific deployments.

To implement Lambda-function access control without requiring that the project team members be granted these IAM privileges, you use the Cloud Canvas `LambdaConfiguration (p. 185)` custom resource. Using the `Metadata.CloudCanvas.FunctionAccess` entries on each of the feature resources to which a Lambda function requires access, the handler for the `LambdaConfiguration` resource creates and configures a role for each Lambda function that allows the function to perform the indicated actions on the resources it requires.

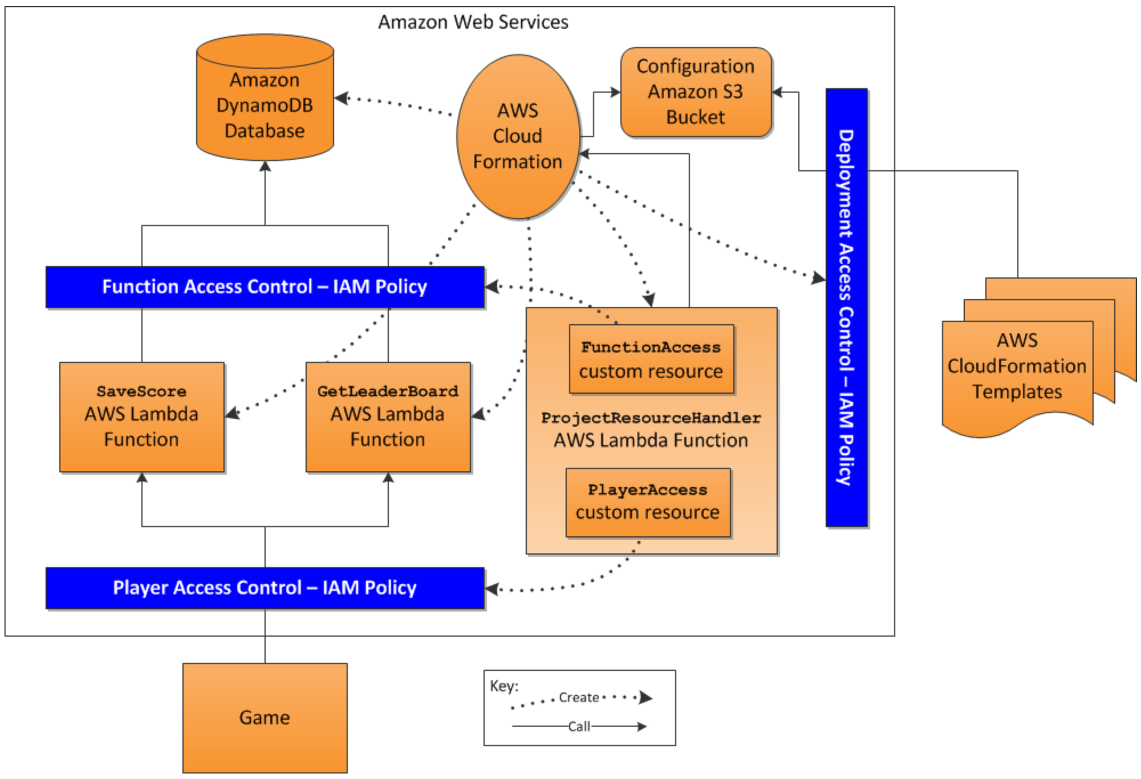
The `Metadata.CloudCanvas.Function` property has the following form:

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "...": {
            "Type": "...",
            "Properties": {
                ...
            },
            "Metadata": {
                "CloudCanvas": {
                    "FunctionAccess": {
                      "Action": [ "{allowed-action-1}", ..., "{allowed-action-
n}" ],
                        "ResourceSuffix": "{resource-suffix}"
                    }
                }
            }
        },
      ...
    }
}
```

The required `Action` property is the same as defined for an IAM policy and is described in detail in the IAM Policy Elements Reference. Note that a single value can be provided instead of a list of values.

The optional `ResourceSuffix` property value is appended to the resource's ARN in the generated policy. This can be used to further restrict access to the resource. For example, for Amazon S3 buckets it can be used to restrict access to objects with matching names. For more information, see Resource in the IAM Policy Elements Reference.

The following diagram illustrates how different elements of Lumberyard access control work together.


Amazon Web Services
Amazon DynamoDB Database
AWS Cloud Formation
Configuration Amazon S3 Bucket
Deployment Access Control – IAM Policy
Function Access Control – IAM Policy
SaveScore AWS Lambda Function
GetLeaderBoard AWS Lambda Function
FunctionAccess custom resource
ProjectResourceHandler AWS Lambda Function
PlayerAccess custom resource
AWS CloudFormation Templates
Player Access Control – IAM Policy
Game
Key:
Create
Call

In the diagrammed example, Lambda functions do things like submit a player's high scores to a DynamoDB database or retrieve the top ten scores from it.

The Player Access Control IAM policy allows the game to call Lambda functions on behalf of the player. In turn, the Function Access Control policy determines the AWS resources that Lambda functions can access (in the example, it's a DynamoDB database). This secure arrangement prevents the player from accessing the DynamoDB database directly and offers the following benefits:

- It enables you to validate the input from the client and remove insecure or unwanted inputs. For example, if a client self-reports an impossibly high or low score, you can reject the unwanted value before it can be written to the database.
- It prevents a customer from trying to access another customer's data.
- It prevents malicious attacks.

To create (and later, update as required) the DynamoDBdatabase, Lambda functions, and access control policies, AWS CloudFormation reads the AWS CloudFormation templates from the Amazon S3 configuration bucket and executes the instructions they contain. AWS CloudFormation reads the `deployment-access-template.json` file and creates a Deployment Access Control IAM policy, which determines which resources AWS CloudFormation can create or update for a particular deployment. This is key in keeping development, test, and live deployments separate and secure from one another.

The templates also use custom resources to implement functionality that AWS CloudFormation by itself cannot perform. In Lumberyard, custom resources are like library functions. For example, the `deployment-access-template.json` file calls the `CognitoIdentityPool` custom resource to create Amazon Cognito identity pools. To create the Function Access Control IAM policy for each Lambda function, the template calls the `LambdaConfiguration` custom resource. The custom resource reads the `FunctionAccess` metadata entries for the particular resources to which the Lambda function should have access and creates the Function Access Control policy that is needed for the current user and deployment.

Similarly, the `feature-template.json` template `PlayerAccess` custom resource is called to create the Player Access Control policy, which determines the Lambda functions and other resources that the game can call and use on behalf of the player.

Player Identity

As described in the preceding section Player Access Control (p. 188), the game must use AWS credentials that grant the desired access when calling AWS APIs (using either the C++ AWS SDK or the AWS flow nodes). Cloud Canvas uses an Amazon Cognito identity pool to get these credentials.

Using a Amazon Cognito identity pool has the benefit of providing the game with a unique identity for each individual player. This identity can be used to associate the player with their saved games, high scores, or any other data stored in DynamoDB tables, Amazon S3buckets, or other locations.

Amazon Cognito identity pools support both unauthenticated and authenticated identities. Unauthenticated identities are associated with a single device such as a PC, tablet, or phone, and have no associated user name or password.

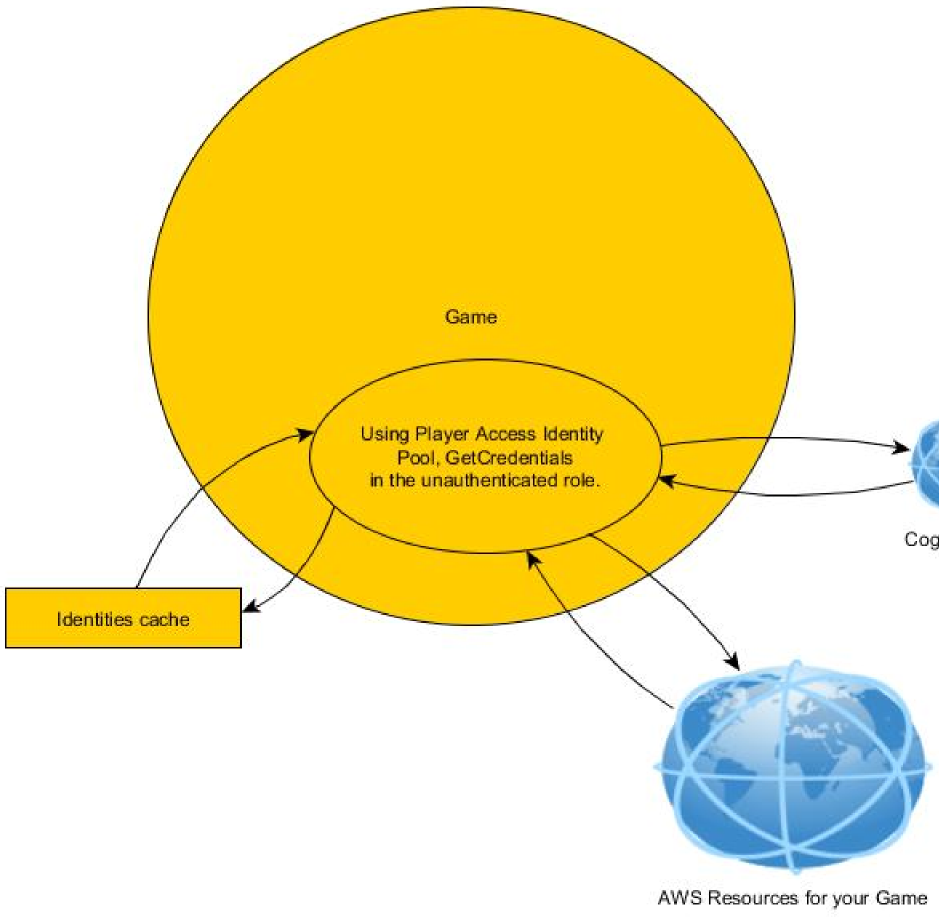
Authenticated identities are associated with the identity of an user as determined by an external identity provider such as Amazon, Facebook, or Google. This allows Amazon Cognito to provide the game with the same player identity everywhere a user plays a game. The user's saved games, high scores, and other data effectively follow the user from device to device.

Amazon Cognito allows a user to start with an unauthenticated identity and then associate that identity with an external identity at a later point in time while preserving the Amazon Cognito-provided identity.

Cloud Canvas supports both anonymous (unauthenticated) and authenticated player identities, but authenticated identity support is more complex and requires additional setup and coding.

Anonymous (Unauthenticated) Player Login

The login process for anonymous (unauthenticated) players is shown in the diagram below:

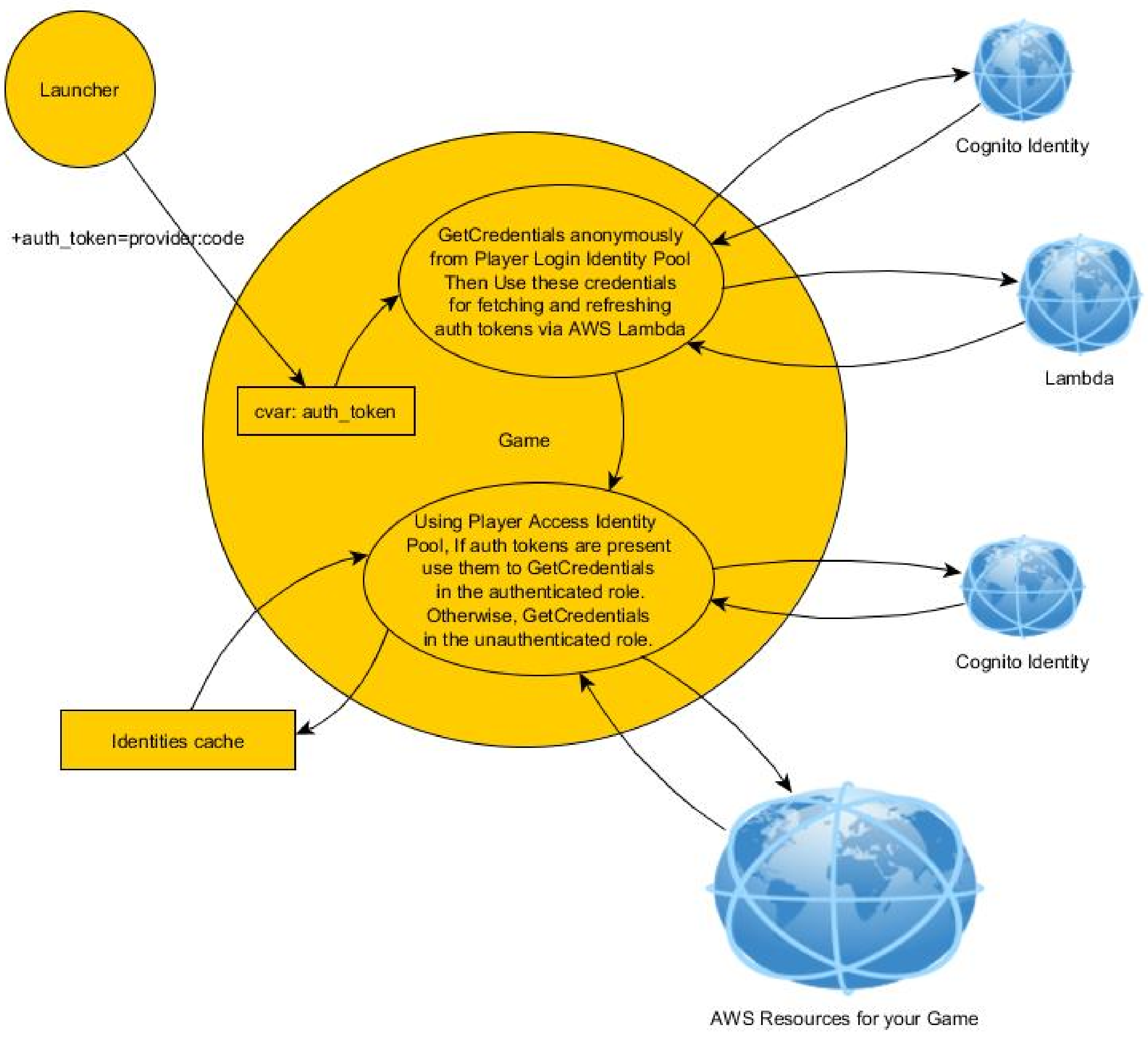

Game
Using Player Access Identity Pool, GetCredentials in the unauthenticated role.
Identities cache
Cognito Identity
AWS Resources for your Game

This process takes place automatically when the Cloud Canvas client configuration system is initialized by calling `gEnv->lmbrAWS->GetClientManager()->ApplyConfiguration()`, or by using a Cloud Canvas `(AWS):Configuration:ApplyConfiguration` Flow Node.

Authenticated Player Login

In order to understand how to use Cloud Canvas to implement authenticated player identities for your game, you must be familiar with Amazon Cognito's Enhanced (Simplified) Authflow. For information, see the article Authentication Flow in the Amazon Cognito Developer Guide.

The login process for authenticated player identities, shown in the diagram that follows, is more complex than the anonymous player login process. This login process requires additional setup beyond what Cloud Canvas provides by default.


Launcher
+auth_token=provider:code
cvar: auth_token
Game
GetCredentials anonymously from Player Login Identity Pool Then Use these credentials for fetching and refreshing auth tokens via AWS Lambda
Using Player Access Identity Pool, If auth tokens are present use them to GetCredentials in the authenticated role. Otherwise, GetCredentials in the unauthenticated role.
Identities cache
Cognito Identity
Lambda
Cognito Identity
AWS Resources for your Game

The authenticated player login process takes place automatically when the Cloud Canvas client configuration system is initialized by calling `gEnv->lmbrAWS->GetClientManager()->ApplyConfiguration()`, or by using a Cloud Canvas `(AWS):Configuration:ApplyConfiguration` flow node.

The presence of the `auth_token` `cvar` triggers the Cloud Canvas authenticated player login flow. If the `cvar` is not set, the anonymous player login process is used. The value of the `cvar` must be a string of the form *`{provider}`*:*`{id}`*, where *`{provider}`* identifies an external identity provider that you have configured for your game (see Configuring External Identity Providers (p. 194) in the section that follows) and *`{id}`* is the player's identity as returned by the login process for that provider.

When `auth_token` is set, Cloud Canvas will pass the provided *`{id}`* value to the `ProjectPlayerAccessTokenExchangeHandler` Lambda function. The Lambda function calls the external provider's API to with the specified ID and receives a value that is passed to Amazon Cognito to get the player's identity and credentials. The calls made by `ProjectPlayerAccessTokenExchangeHandler` use application IDs and the secret values you provide as part of the external identity provider configuration process.

As shown in the diagram above, Cloud Canvas uses one Amazon Cognito identity pool to get the credentials used to invoke the `ProjectPlayerAccessTokenExchangeHandler` and a different Amazon

Cognito identity pool to get the credentials used to access the rest of your game's resources. This is required because access `ProjectPlayerAccessTokenExchangeHandler` is always anonymous.

All the code that implements the authenticated login flow can be found in the `{root}\Code\CryEngine\LmbrAWS\Configuration` directory. A description of the files follows.

- `ClientManagerImpl.*` – Configures the game's AWS clients to use the `TokenRetrievingPersistentIdentityProvider` identity provider.
- `ResourceManagementLambdaBasedTokenRetrievalStrategy.*` – implements the token exchange process that calls the `ProjectPlayerAccessTokenExchangeHandler` Lambda function.
- `TokenRetrievingPersistentIdentityProvider.*` – An implementation of the `PersistentCognitoIdentityProvider` interface defined in the AWS SDK that uses `ResourceManagementLambdaBasedTokenRetrievalStrategy` instances to implement the token exchange process.

Configuring External Identity Providers

Cloud Canvas does not automate the process of retrieving an auth code from an external identity provider and setting the `auth_token cvar`. This is your responsibility as a game developer. Following are some possible implementation methods:

- On a PC, you can have your identity provider redirect its URI to a static web page that redirects the user to a custom URI. You can use the custom URI to launch the game and pass the auth code as a command line argument (for example, `yourGame.exe +auth_token=provider:code`). Cloud Canvas detects this command line argument and logs the user into your game. This only has to be done once since the auth tokens are cached locally.
- You can have your game retrieve the auth code itself (but for many external identity providers, this may require using an embedded web browser). After retrieving the auth code, you can call `gEnv->lmbrAWS->->GetClientManager()->Login(providerName, code)`, or just set the `cvar auth_token`.
- If you have a launcher for your game, you can embed a web browser window in the launcher to allow the player to log in to the external identity provider. You can then retrieve the auth code and launch the game by using the `+auth_token=provider:code` parameter.

External identity providers are configured using the `lmbr_aws` add-login-provider (p. 172), update-login-provider (p. 181), and remove-login-provider (p. 179) commands. These commands save the configuration in a `/player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

Note
You must run `lmbr_aws update-project` after running `add-login-provider`, `update-login-provider`, or `remove-login-provider` so that the PlayerAccessIdentityPool Resource (p. 162) configuration will be updated to reflect the change.

Automatic Token Refresh

When using Amazon Cognito with external identity providers, it is necessary to periodically refresh the token from that provider and then get updated credentials for that token from Amazon Cognito. Cloud Canvas performs this token refresh process automatically by using the `ProjectPlayerAccessTokenExchangeHandler` Lambda function.

AWS Client Configuration

Access to AWS services from C++ is controlled through AWS client objects. Cloud Canvas provides a layer of abstraction on top of the AWS clients given by the SDK. This enables you to easily apply configuration changes across client objects, including those being used internally by Cloud Canvas flow nodes. These changes can be made through C++ or with the provided flow nodes.

Configuring AWS Flow Nodes

Lumberyard provides flow nodes that perform various AWS operations, such as invoking a Lambda function or reading data from a DynamoDB table. Each flow node has a port that identifies the AWS resource to operate on. For example, DynamoDB nodes have a `TableName` port that identifies a DynamoDB table.

By default, the AWS resource uses the value you specify for the name. You can use the default client settings to configure the client that accesses the resource.

To vary the resource name based on game settings

1. In the default client settings, add a parameter reference in the resource name.
2. Use the `Cloud Canvas (AWS):Configuration:SetConfigurationVariable` flow node to set the value that replaces the parameter reference in the resource name.

For example, if you aren't using Resource Deployments but still want to manage resources in different stages, you can use **`"DailyGift_$stage$"`** for your table name and set the `"$stage$"` parameter value to **`"test"`** or **`"release"`** based on the conditions you choose.

If you want to use different client settings (for example, timeouts) for different resources, you can use the C++ API to set up named configurations. When the resource name provided to the flow node matches a named configuration, the configuration is used to set up the client to access the resource.

The configuration-related flows are as follows:

- `ConfigureProxy` – Configures the HTTP proxy host, port, user, and password used for AWS service calls.
- `SetConfigurationVariable` – Sets the parameter value for the AWS resource.
- `GetConfigurationVariableValue` – Replaces the `"$param$"` substrings in a string with the current parameter value or `""` if there is no current value.
- `SetDefaultRegion` – Sets the region for all AWS clients in the current project.
- `ApplyConfiguration` – Applies AWS configuration settings set by the other configuration flow nodes. This allows you to apply multiple changes at once to prevent changes from being applied out of the desired order.

Configuring Using C++

The Client Manager implementation and the AWS flow node implementations are provided through the Cloud Canvas Gem. To use the Client Manager, add the **Cloud Canvas Gem** to your game project by using the Project Configurator. To get the Client Manager in code, call `gEnv->pLmbrAWS->GetClientManager()`. This will return an interface called `IClientManager`.

The Client Manager maintains a set of parameter name to parameter value mappings. You can access these mappings by using the object returned by the `GetConfigurationParameters` method. You can then insert parameter values into certain configuration settings by using `"$param-name$"` substrings.

If the parameter value contains `"$param-name$"` substrings, these are also replaced with the corresponding parameter value.

AWS client configurations are based on the properties of the object returned by the Client Manager's `GetDefaultClientSettings` method. You can specify named overrides for these settings by using the collection object returned by the `GetClientSettingsOverridesCollection` method. Overrides can aggregate other overrides by name, allowing common configuration settings to be specified once and then reused. These names can include `"$param-name$"` values, which are replaced as described above. If the resource name for an AWS flow node matches the name of a client settings overrides collection entry, those overrides are used by the client to access the resource.

Service-specific configurations are provided through the following Client Manager methods: `GetCognitoIdentityManager`, `GetDynamoDBManager`, `GetIAMManager`, `GetLambdaManager`, `GetS3Manager`, `GetSNSManager`, `GetSQSManager`, and `GetSTSManager`.

Each of these service-specific managers provide the following:

- A `GetDefaultClientSettingsOverrides` method that returns an object that can be used to define the default settings for all of a particular service's clients.
- Methods that allow resources to be configured by name.

For example, the DynamoDB manager has a `GetTableClientSettingsCollection` method that you can use to configure settings (the configuration name and resource name associated with the configuration) for a specific DynamoDB table. The resource name can include `"$param-name$"` values, which are replaced as described above.

In an AWS flow node, you can use the configuration name as the resource name in order to use that configuration. In this case, the configuration name acts as an alias for the actual resource name. For example, a node could specify **`"DailyGift"`** as the table name and the resource configuration `"DailyGift"` could specify **`"foo-bar-$stage$"`** as the resource name.

If you use the name **`"DailyGift_$stage$"`** in a flow node and do not define a resource configuration for the name **`"DailyGift_$stage$"`**, a default configuration is created that uses **`"DailyGift_$stage$"`** for both the configuration name and resource name.

Using Configured Clients from C++

The Client Manager methods (`GetCognitoIdentityManager`, `GetDynamoDBManager`, `GetIAMManager`, `GetLambdaManager`, `GetS3Manager`, `GetSNSManager`, `GetSQSManager`, and `GetSTSManager`) return objects that provide methods for creating clients configured with the settings described above.

The `CreateManagedClient` method returns a client object that is configured using the client settings overrides identified by the provided name. This object can be passed around by value. The client's methods are accessed using the `->` operator.

You can use the resource-specific `CreateXClient` methods (for example, `CreateTableClient` for a DynamoDB table resource) to configure a client to access a specific resource. The returned object has a method for retrieving the resource name (such as `client.GetTableName`), while the client methods are accessed using the `->` operator (e.g. `client->GetItem(...)`).

Important
You can create clients in an unconfigured state. Flow nodes and other game components can create clients before the configuration for the clients is fully initialized. For example, the Amazon Cognito credentials provider and configuration parameters may need to be initialized by flow nodes that are triggered at game start, before other flow nodes or components can access AWS services. Because of this, you must trigger client configuration explicitly once the settings are confirmed complete.

You must call the Client Manager's `ApplyConfiguration` method at least once before clients are ready for use. After you make configuration changes, you must call the method again for those changes to take effect. You can use the `IsReady` method (`client.IsReady(), not client->IsReady()`) to determine if a client has been configured.

Don't Die Sample Project

This sample project serves as an example to illustrate the use of the AWS Cloud Canvas Resource Management system and AWS Lambda for a game. *Don't Die* also uses Project Configurator, which is a standalone application included with Lumberyard which which you can specify to the Waf Build system which game project and assets (Gems) to include in a build.

AWS resources used in the *Don't Die* project may be subject to separate charges and additional terms. There is a free tier for all the AWS services used in this project. See the end of this topic to learn more about AWS services used here.

Setup

Setting up the sample project involves a few tasks.

Initializing the AWS Project Stack

All AWS resources associated with *Don't Die*, such as DynamoDB tables, S3 buckets, and Lambda, are created through the Cloud Canvas Resource Management system. Any cloud-connected features of *Don't Die*, such as High Score and Daily Gift, will not work before setup is completed. In order to set up these AWS resources, you must meet several prerequisites:

- Have an AWS account with administrator permissions credentials.
- Ensure that **SamplesProject** is selected in Project Configurator. This points the AWS Resource Management System to `\SamplesProject\AWS`, which contains all the files, templates, and project code needed to create the needed AWS resources.
- Ensure that the `SamplesProject\AWS\project-settings.json` file is writable. During the resource set up process, the Amazon Resource Name (ARN) for the AWS CloudFormation stack is added into this file.
- Select the region where the AWS resources will live. *Don't Die* relies on a service called Cognito for player identity, which is currently only supported in the following regions: US East (N. Virginia) (us-east-1), EU (Ireland) (eu-west-1), and Asia Pacific (Tokyo) (ap-northeast-1). This example uses US East (N. Virginia), but the other two are valid options as well.

To initialize the project stack

- From a command line prompt, go to the root Lumberyard folder, and enter the following (assumes US East region): `lmbr_aws initialize-project --region us-east-1`.

Setting up Event Sources

With AWS, you can execute a Lambda function whenever certain events occur, such as editing of a DynamoDB table, or adding to an S3 bucket. This is handled by setting up an event source on the Lambda function that listens for an event. For now, event sources are added to a project manually.

The *Don't Die* project relies on three event sources. Wwo game systems in *Don't Die* rely on a table that contains cached data. This cached data is updated when one of the following occurs:

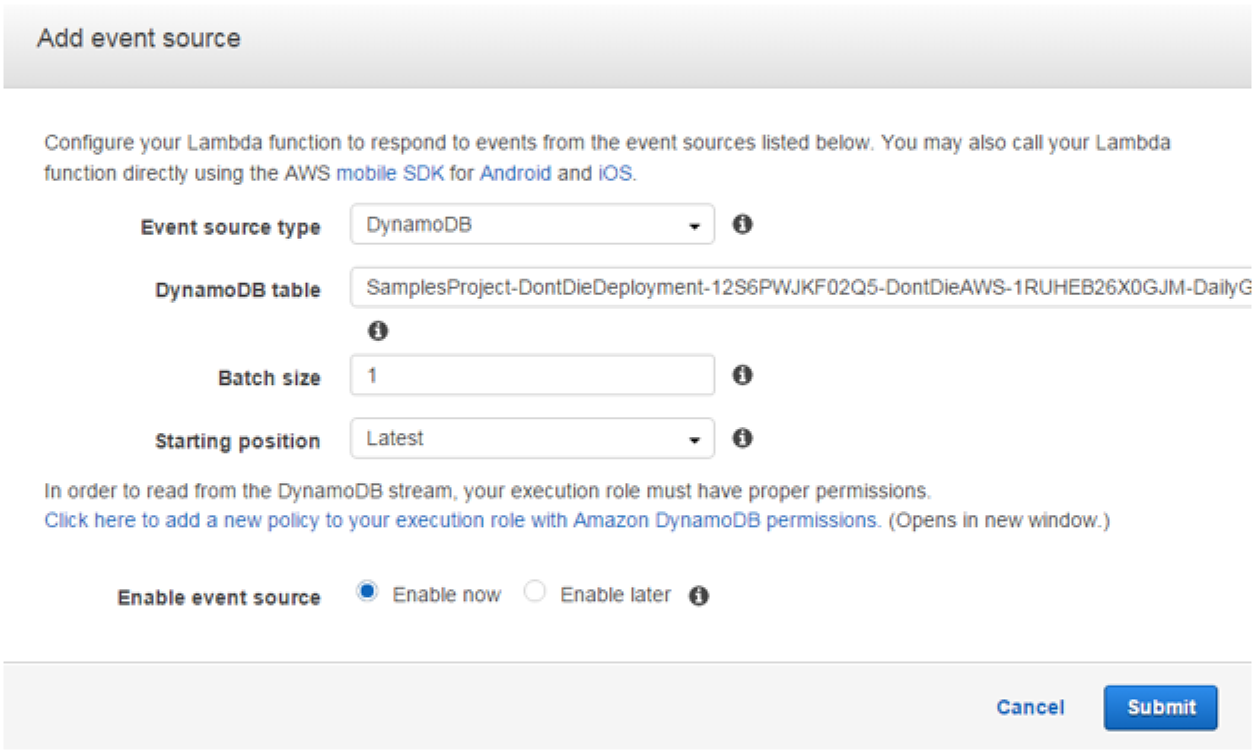
- The **Daily Gift** table changes
- The **Message of the Day** table changes
- Start of a new day

To detect when one of these has occurred, a separate event source is used. Ensure that your currently selected region matches the region where you set up the project.

To select the Lambda for event sources

1. Open the AWS console.
2. On the **Amazon Web Services** page, click **Lambda**.
3. In the **Filter** dialog, enter **`DontDieUpdateGameDataLU`**
4. Select the Lambda function that appears.

Note
If you set up the project stack multiple times, make sure that you select the Lambda function that has the matching stack name.


Add event source

Configure your Lambda function to respond to events from the event sources listed below. You may also call your Lambda function directly using the AWS mobile SDK for Android and iOS.

Event source type: DynamoDB

DynamoDB table: SamplesProject-DontDieDeployment-12S6PWJKF02Q5-DontDieAWS-1RUHEB26X0GJM-DailyG

Batch size: 1

Starting position: Latest

In order to read from the DynamoDB stream, your execution role must have proper permissions.
Click here to add a new policy to your execution role with Amazon DynamoDB permissions. (Opens in new window.)

Enable event source: [✓ Enable now] Enable later

Next, perform the following steps to set up an event source for the Daily Gift feature:

To add the Daily Gift event source

1. On the Lambda **Settings** page, click the **Event Sources** tab, and then click **Add event source**.
2. For **Event source type**, select **DynamoDB**.
3. In the **DynamoDB** table field, select the drop down, and filter for **DailyGiftTable**. Select the one that corresponds to your project.
4. Set **BatchSize** to 1. This controls how many updates can be batched together and sent to the Lambda function.

5. Set **Starting position** to **Latest**. This means that no changes made previous to the creation of this event source will be sent to the Lambda and only recent changes will be passed along. If you have made changes to the table before adding this event source, delete its cache by selecting **GameDataLUTable** and deleting **TodaysGift**.
6. Select **Enable now**.
7. Click **Submit**.

A new event source should now show up in your list of event sources. Each time someone edits the **Daily Gift** table, this Lambda will now be executed.

Next, setup the **Message of the Day** event source. Click **Add event source** again, like before:

To add the Message of the Day event source

1. Click **Add event source**.
2. For **Event source type**, select **DynamoDB**.
3. For **DynamoDB**, use the drop-down to filter for **MessageOfTheDay**. Select the one that corresponds to your project.
4. Set **BatchSize** to **1**.
5. Set **Starting position** to **Latest**.
6. Select **Enable now**
7. Click **Submit**

There should now be two event sources listed. There may be a delay between setting up the event source and when the Lambda function actually begins to receive updates from DynamoDB. Therefore, wait a few minutes after creating the above event sources before editing the **Daily Gift** or **Message of the Day** tables.

The third event source is different. Instead of pointing to another AWS resource, it acts as a timer, executing the Lambda function every day at midnight. For *Don't Die*, the time zone is in a PST instead of GMT for convenience. (8:00 GMT = PST).

To add the timer event source

1. Click **Add event source**.
2. For **Event source type**, select **CloudWatch Events - Schedule**.
3. Enter **`AtMidnight`** as the name. The Description field can be left blank
4. For **Schedule expression**, enter the following: `cron(0 8 * * ? *)`
5. Select **Enable now**.
6. Click **Submit**.

Viewing Lambda Code in Visual Studio

The Lambda source for *Don't Die* can be viewed using Visual Studio 2013. However, there are a couple of tools that need to be installed first:

- AWS Toolkit for Visual Studio
- Node JS Tools for Visual Studio 2013

Acquiring the Mappings File

All of the Lambda code refers to AWS resources by friendly names like "High Score Table." Lambda functions use a mappings file to translate friendly names to the physical names of the actual AWS resources in your account. This file is generated by the AWS Resource Management System but is only inserted in the Lambda function right before the system uploads the code to AWS. In order to run the Lambda functions from Visual Studio, copy the mappings file locally, as follows:

To acquire the mappings file

1. Open the AWS console and find the **Lambda** section.
2. Filter for **DontDie** and select either of the Lambda functions that appear. One should contain the text `DontDieAWSMain` and the other `DontDieUpdateGameDataLU`.
3. Click **Actions** and select **Download function code**.
4. Open the downloaded `.zip` file and extract the `CloudCanvas/settings.js` file to *`lumberyard_root`*`\SamplesProject\AWS\feature\DontDieAWS\feature-code\CloudCanvas`.

Once this file is downloaded and available locally, Visual Studio can find the correct resources for the Lambda function.

Lambda Code Overview

Don't Die has two Lambda function but but only one Visual Studio project. This is because while both Lambda functions contain the same code, they have different entry points. Within the Visual Studio project, under the **apps** folder, there are two other folders - `\dont-die-main` and `\dont-die-update-game-data-lu`. Both of these folders correspond to a Lambda for*Don't Die*.

The `dont-die-main` Lambda function is used for everything in the game. It executes commands such as `start-game`, `end-game`, and `get-high-score-table`) that are sent to it from the client, and then sends back the result.

The `dont-die-update-game-data-lu` Lambda function, ("lu" here means "lookup") is executed by the event sources that were set up in the previous section. This Lambda function keeps up to date a DynamoDB table that contains cached data from other tables.

Both Lambda folders contain a `_sampleEvent.json`, `_testdriver.js`, and a `app.js` file. These files are not uploaded to AWS (this is controlled by the `.ignore` file). They for local testing only.

The `_sampleEvent.json` file contains test data that acts as stand-in for data that the client sends. The `_testdriver.js` file executes the Lambda code locally, acting as the part of AWS that calls the Lambda function. Both of these files are generated by the AWS toolkit, although they have been somewhat modified.

To test a specific Lambda function, right-click the applicable `_testdriver.js` file, select **Set as Node.js Startup File**, and click **Start Debugging**.

Both Lambda functions exhibit the same general behavior:

- Creates a `dontdie` global that is shared across all concurrently running Lambda functions.
- Creates a `GameContext` that represents that particular execution of the Lambda function.
- Each `GameContext` in turn spins up needed game systems, based on flags given to it. These flags may also turn off functionality inside of a game system.
- Performs any logic to manipulate game state. In the case of the main Lambda, this means executing the commands passed in by the client. Commands are currently registered by each game system.
- Once finished, calls `FinishGameContext`, which will cause `Finish` to be called on each of the game systems. This gives each game system a chance to respond to any changes to state that have been made. For example, if player data was changed, it will be saved to DynamoDB at this point.

Deleting the AWS Project Stack

Don't Die creates one S3 bucket. Before you can delete the project stack, ensure this S3 bucket is empty. Otherwise, the AWS Resource Management system cannot delete the S3 bucket, which blocks it from deleting the project stack.

To empty an S3 bucket

1. Open the AWS Management Console and click **S3**.
2. Find the **samplesproject** bucket that has "**mainbucket**" somewhere in the title, and click that bucket.
3. Select each item in the bucket, and then click **Actions**, **Delete**.

Now that the S3 bucket is empty, you can proceed with deleting the project stack.

To delete the project stack

1. Open the `project-settings.json` file.
2. Look for the key **Stack ID**. The value of **Stack ID** is the Amazon Resource Name that points to your AWS CloudFormation stack, which keeps track of all the AWS resources for the game project.
3. Copy this string, and then remove the key-value pair from the file.
4. From a command line prompt, go to the Lumberyard root folder, and then type the following command: `lmbr_aws delete-project-stack --project-stack-id` ***`stack_id`***.

If you accidentally delete the StackId from the project settings file, you can retreive it in the AWS Management Console:

To retrieve the Stack ID

1. Open the AWS Management Console and click **CloudFormation**.
2. Filter for the **SamplesProject** name, locate it, and then click the stack name that matches it exactly.
3. At the bottom of the screen, in the **Overview** tab, look for a table row titled **Stack ID**. Copy the Stack ID value shown and paste it into the command line for Step 4 (***`stack_id`***).

AWS Services Used

The *Don't Die* sample project implements game back-end features on AWS. By default, this project uses the services listed in the table below. When you use certain features in the sample project, you are using the AWS services that power them. You can add additional services by customizing the templates or writing your own templates.

When you initialize the *Don't Die* sample project, you are prompted to deploy AWS services to your account using the included AWS CloudFormation templates.

There is no additional charge for using Cloud Canvas. AWS resources you use for *Don't Die* may be subject to separate charges and additional terms. You pay for AWS resources created using Cloud Canvas, such as Lambda functions, DynamoDB tables, and IAM in the same manner as if you created them manually. You only pay for what you use, as you use it; there are no minimum fees and no required upfront commitments, and most services include a free tier.

AWS Services Table

Feature	AWS Services Used
Setup	AWS CloudFormation, DynamoDB, DynamoDB, Amazon S3, Amazon Cognito, IAM
Message of the Day example	AWS CloudFormation, Lambda, DynamoDB,
Achievements example	AWS CloudFormation, Lambda, DynamoDB,
High Scores example	AWS CloudFormation, Lambda, DynamoDB, S3
Daily Gift example	AWS CloudFormation, Lambda, DynamoDB,
Item Manager example	AWS CloudFormation, Lambda, DynamoDB,
Mission example	AWS CloudFormation, Lambda, DynamoDB,

Controller Devices and Game Input

This section provides insight into Lumberyard's support for input devices, including information on setting up controls and action maps.

This section includes the following topics:

Action Maps

The Action Map Manager provides a high-level interface to handle input controls inside a game. The Action Map system is implemented in Lumberyard, and can be used directly by any code inside Lumberyard or the GameDLL.

Initializing the Action Map Manager

The Action Map Manager is initialized when Lumberyard is initialized. Your game must specify the path for the file `defaultProfile.xml` (by default, the path is `Game/Libs/Config/defaultProfile.xml`). You can do this by passing the path to the manager. For example:

```
IActionMapManager* pActionMapManager = m_pFramework->GetIActionMapManager();
if (pActionMapManager)
{
    pActionMapManager->InitActionMaps(filename);
}
```

Upon initialization, the Action Map Manager clears all existing initialized maps, filters, and controller layouts.

Action Map Manager Events

If you have other systems in place that should know about action map events, you can subscribe to Action Map Manager events using the interface `IActionMapEventListener`.

The following events are available:

- `eActionMapManagerEvent_ActionMapsInitialized` – Action map definitions have been successfully initialized.
- `eActionMapManagerEvent_DefaultActionEntityChanged` – Default action entity has been changed (Manager will automatically assign new action maps to this entity).
- `eActionMapManagerEvent_FilterStatusChanged` – An existing filter has been enabled/disabled.
- `eActionMapManagerEvent_ActionMapStatusChanged` – An existing action map has been enabled/disabled.

Receiving Actions During Runtime

You can enable the feature that allows action maps to receive actions during runtime. Use the following code to enable or disable an action map during runtime:

```
pActionMapMan->EnableActionMap("default", true);
```

To receive actions, implement the `IActionListener` interface in a class.

CryInput

The main purpose of CryInput is to provide an abstraction that obtains input and status from various input devices such as a keyboard, mouse, joystick, and so on.

It also supports sending feedback events back to input devices—for example, in the form of force feedback events.

The common interfaces for the input system can be found in `IInput.h`, in the CryCommon project.

IInput

`IInput` is the main interface of the input system. An instance implementing this interface is created automatically during system initialization in the `InitInput` function (`InitSystem.cpp` in CrySystem, see also `CryInput.cpp` in CryInput).

Only one instance of this interface is created. CrySystem also manages the update and shutdown of the input system.

This `IInput` instance is stored in the `SSystemGlobalEnvironment` structure gEnv. You can access it through `gEnv->pInput` or, alternatively, through the system interface by `GetISystem()->GetIInput()`. Access through the `gEnv` variable is the most commonly used method.

IInputEventListener

A common use case within the input system is to create listener classes in other modules (for example, CryGame) by inheriting from `IInputEventListener` and registering/unregistering the listener class with the input system for notifications of input events.

For example, the Action Map System registers itself as an input listener and forwards game events only for the keys defined in the profile configuration files to further abstract the player input from device to the game.

SInputEvent

`SInputEvent` encapsulates information that is created by any input device and received by all input event listeners.

IInputDevice

Input devices normally relate directly to physical input devices such as a joypad, mouse, keyboard, and so on. To create a new input device, you must implement all functions in the `IInputDevice` interface and register an instance of it with the Input System using the `AddInputDevice` function.

The `Init` function is called when registering the `IInputDevice` with the Input System; it is not necessary to manually call it when creating the input devices.

The `Update` function is called at every update of the Input System—this is generally where the state of the device should be checked/updated and the Input Events generated and forwarded to the Input System.

It is common for input devices to create and store a list in SInputSymbol of each symbol the input device is able to generate in the Init function. Then, in the update function, the symbols for the buttons/axes that changed are looked up and used (via their AssignTo function) to fill in most of the information needed for the events, which are then forwarded to the input system.

Example:

```
// function from CInputDevice (accessible only within CryInput)
MapSymbol(...)
{
     SInputSymbol\* pSymbol = new SInputSymbol( deviceSpecificId, keyId, name,
 type );
     pSymbol->user = user;
     pSymbol->deviceId = m_deviceId;
     m_idToInfo\[ keyId \] = pSymbol;
     m_devSpecIdToSymbol\[ deviceSpecificId \] = pSymbol;
     m_nameToId\[ name \] = deviceSpecificId;
     m_nameToInfo\[ name \] = pSymbol;

     return pSymbol;
}
bool CMyKeyboardInputDevice::Init()
{
      ...
     //CreateDeviceEtc();
      ...
     m_symbols\[ DIK_1 \] = MapSymbol( DIK_1, eKI_1, "1" );
     m_symbols\[ DIK_2 \] = MapSymbol( DIK_2, eKI_2, "2" );
      ...
}
void CMyKeyboardInputDevice::Update( ... )
{
     // Acquire device if necessary
      ...
    // Will probably want to check for all keys, so the following section might
 be part of a loop
     SInputSymbol\* pSymbol = m_symbols\[ deviceKeyId \];
      ...
     // check if state changed
    ...
    // This is an example for, when pressed, see ChangeEvent function for axis
 type symbols
    pSymbol->PressEvent( true );

    SInputEvent event;
    pSymbol->AssignTo( event, modifiers );

    gEnv->pInput->PostInputEvent( event );
}
```

To forward events to the input system so that event listeners can receive them, use the `PostInputEvent` function from `IInput`.

If adding your input device to CryInput, it may be useful to inherit directly from `CInputDevice`, as it already provides a generic implementation for most functions in IInputDevice.

Note
This file is included with the full source of CryEngine and is not available in the FreeSDK or GameCodeOnly solutions. For these licenses please derive from IInputDevice directly.

Setting Up Controls and Action Maps

This section describes how to create and modify action maps to customize the controls to the needs of your game.

Action map profiles for all supported platforms are located in `Game\Libs\Config\Profile\DefaultProfile.xml`. This default XML file organizes controls into the following sections, each of which is controlled by its own action map:

- multiplayer
- singleplayer
- debug
- flycam
- default
- player
- vehicle
- land vehicle
- sea vehicle
- helicopter

Each action map can be enabled or disabled during runtime from Flow Graph, in Lua scripts, or in C++ code.

See the topic Default Controller Mapping (p. 209) for an overview of the controls in the SDK package.

Action Maps

An action map is a set of key/button mappings for a particular game mode. For example, there is an `<actionmap>` section for helicopter controls called "Helicopter", which means that everything inside that section consists of key and button bindings that apply only when flying a helicopter. To change your common in-game bindings, go to the section starting with `<actionmap name="default">`. There are also sections for multiplayer-specific bindings and, of course, any other vehicles or modes you need.

The following is an overview of a standard action map, in this case the standard debug one:

```
<actionmap name="debug" version="22">
 <!-- debug keys – move to debug when we can switch devmode-->
  <action name="flymode" onPress="1" noModifiers="1" keyboard="f3" />
  <action name="godmode" onPress="1" noModifiers="1" keyboard="f4" />
  <action name="toggleaidebugdraw" onPress="1" noModifiers="1" keyboard="f11"
/>
  <action name="togglepdrawhelpers" onPress="1" noModifiers="1" keyboard="f10"
 />
  <action name="ulammo" onPress="1" noModifiers="1" keyboard="np_2" />
  <action name="debug" onPress="1" keyboard="7" />
  <action name="thirdperson" onPress="1" noModifiers="1" keyboard="f1" />
  <!-- debug keys – end -->
 </actionmap>
```

Versioning

```
<actionmap name="debug" version="22">
```

When the version value is incremented, Lumberyard ensures that the user profile receives the newly updated action map. This is quite useful when deploying new actions in a patch of a game that is already released. If the version stays the same, changes or additions to the action maps are not propagated to the user profile.

Activation Modes

The following activation modes are available:

- onPress – The action key is pressed
- onRelease – The action key is released
- onHold – The action key is held
- always – Permanently activated

The activation mode is passed to action listeners and identified by the corresponding Lua constant:

- eAAM_OnPress
- eAAM_OnRelease
- eAAM_OnHold
- eAAM_Always

Modifiers available:

- retriggerable
- holdTriggerDelay
- holdRepeatDelay
- noModifiers – Action takes place only if no **Ctrl**, **Shift**, **Alt**, or **Win** keys are pressed
- consoleCmd – Action corresponds to a console command
- pressDelayPriority

- pressTriggerDelay
- pressTriggerDelayRepeatOverride
- inputsToBlock – Specify the input actions to block here
- inputBlockTime – Time to block the specified input action

Action Filters

You can now also define action filters directly in your defaultProfile.xml. The following attributes are available:

- name – How the filter will be identified
- type (actionFail, actionPass) – Specifies whether the filter will let an action fail or pass

```
<actionfilter name="no_move" type="actionFail">
 <!-- actions that should be filtered -->
  <action name="crouch"/>
  <action name="jump"/>
  <action name="moveleft"/>
  <action name="moveright"/>
  <action name="moveforward"/>
  <action name="moveback"/>
  <action name="sprint"/>
  <action name="xi_movey"/>
  <action name="xi_movex"/>
 <!-- actions end -->
</actionfilter>
```

Controller Layouts

Links to the different controller layouts can also be stored in this file:

```
<controllerlayouts>
 <layout name="Layout 1" file="buttonlayout_alt.xml"/>
 <layout name="Layout 2" file="buttonlayout_alt2.xml"/>
 <layout name="Layout 3" file="buttonlayout_lefty.xml"/>
 <layout name="Layout 4" file="buttonlayout_lefty2.xml"/>
</controllerlayouts>
```

Note
The "file" attribute links to a file stored in "libs/config/controller/" by default.

Working with Action Maps During Runtime

In Lumberyard, you can use the console command `i_reloadActionMaps` to re-initialize the defined values. The ActionMapManager sends an event to all its listeners to synchronize the values throughout the engine. If you're using a separate GameActions file like GameSDK, make sure this class will receive the update to re-initialize the actions/filters in place. Keep in mind that it's not possible to define action maps, filters, or controller layouts with the same name in multiple places (for example, action filter no_move defined in `defaultProfile.xml` and the `GameActions` file).

To handle actions during runtime, you can use flow graphs or Lua scripts.

- Flow Graph – Input nodes can be used to handle actions. Only digital inputs can be handled from a flow graph. For more information, see Flow Graph System in the Amazon Lumberyard User Guide.
- Lua script – While actions are usually not intended to be received directly by scripts, it is possible to interact with the Action Map Manager from Lua.

Default Controller Mapping

The default mapping for input on the PC is shown in the following table. To reconfigure the controls for your game, follow the instructions in Setting Up Controls and Action Maps (p. 206) and Action Maps (p. 203).

Player Action	PC
Player Movement	W, A, S, D
Player Aim	Mouse XY
Jump	Spacebar
Sprint	Shift
Crouch	C
Slide (when sprinting)	C
Fire	Mouse 1
Zoom	Mouse 2
Melee	V
Fire Mode	2
Reload	R
Use	F
Toggle Weapon	1
Toggle Explosive	3
Toggle Binoculars	B
Toggle Light (attachment)	L
Third Person Camera	F1

Vehicle Action	PC
Accelerate	W
Boost	Shift
Brake/Reverse	S
Handbrake	Spacebar
Steer	A/D
Look	Mouse XY

Vehicle Action	PC
Horn	H
Fire	Mouse 1
Change Seat	C
Headlights	L

Helicopter Action	PC
Ascend	W
Descend	S
Roll Left	A
Roll Right	D
Yaw Left	Mouse X (left)
Yaw Right	Mouse X (right)
Pitch Up	Mouse Y (up)
Pitch Down	Mouse Y (down)

Multiplayer Action	PC
Show Scoreboard	TAB

Key Naming Conventions

This page lists some of the name conventions used for action maps.

Key Gestures

Letters	"a" - "z"
Numbers	"1" - "0"
Arrows	"up", "down", "left", "right"
Function keys	"f1" - "f15"
Numpad	"np_1" - "np_0", "numlock", "np_divide", "np_multiply", "np_subtract", "np_add", "np_enter", "np_period"
Esc	"escape"
~	"tilde"
Tab	"tab"
CapsLock	"capslock"
Shift	"lshift", "rshift"

Ctrl	"lctrl", "rctrl"
Alt	"lalt", "ralt"
spacebar	"space"
-	"minus"
=	"equals"
Backspace	"backspace"
[ ]	"lbracket", "rbracket"
"\"	"backslash"
;	"semicolon"
'	"apostrophe"
Enter	"enter"
,	"comma"
.	"period"
/	"slash"
Home	"home"
End	"end"
Delete	"delete"
PageUp	"pgup"
PageDown	"pgdn"
Insert	"insert"
ScrollLock	"scrolllock"
PrintScreen	"print"
Pause/Break	"pause"

Mouse Gestures

Left/primary mouse button	"mouse1"
Right/secondary mouse button	"mouse2"
Mouse wheel up	"mwheel_up"
Mouse wheel down	"mwheel_down"
New position along x-axis	"maxis_x"
New position along y-axis	"maxis_y"

CryCommon

The `Code\CryCommon` directory is the central directory for all the engine interfaces (as well as some commonly used code stored there to encourage reuse).

This section includes the following topics:

CryExtension

The complexity of Lumberyard can be challenging to both newcomers and experienced users who want to understand, configure, run, and extend it. Refactoring Lumberyard into extensions makes it easier to manage. Existing features can be unplugged (at least to some degree), replaced, or customized, and new features added. Extensions can consolidate code for a single feature in one location. This avoids having to implement a feature piecemeal across a number of the engine's base modules. Refactoring into extensions can also make the system more understandable at a high level.

Lumberyard's extension framework is loosely based on some fundamental concepts found in Microsoft's Component Object Model (COM). The framework defines two base interfaces that each extension needs to implement, namely `ICryUnknown` and `ICryFactory`. These are similar to COM's `IUnknown` and `IClassFactory`. The interfaces serve as a base to instantiate extensions, allow interface type casting, and enable query and exposure functionality.

The framework utilizes the concept of shared pointers and is implemented in a way to enforce their consistent usage to help reduce the chance of resource leaks. A set of C++ templates wrapped in a few macros is provided as Glue Code Macros (p. 218) that encourage engine refactoring into extensions. The glue code efficiently implements all base services and registers extensions within the engine. Additionally, a few helper functions implement type-safe casting of interface pointers, querying the IDs of extension interfaces, and convenient instantiation of extension classes. Hence, repetitive writing of tedious boilerplate code is unnecessary, and the potential for introducing bugs is reduced. An example is provided in the section Using Glue Code (p. 227). If the provided glue code is not applicable, then you must implement the interfaces and base services manually, as described in the section Without Using Glue Code (p. 229).

Clients access extensions through a system wide factory registry. The registry allows specific extension classes to be searched by either name or ID, and extensions to be iterated by using an interface ID.

Composites

The framework allows extensions to expose certain internal objects that they aggregate or are composed of. These so called *composites* are extensions themselves because they inherit from `ICryUnknown`. Composites allow you to reuse desired properties like type information at runtime for safe casting and loose coupling.

Shared and raw interface pointers

Although the framework was designed and implemented to utilize shared pointers and enforce their usage in order to reduce the possibility of resource leaks, raw interface pointers can still be acquired. Therefore, care needs to be taken to prevent re-wrapping those raw interface pointers in shared pointer objects. If the original shared pointer object is not passed during construction so that its internal reference counter can be referred to, the consistency of reference counting will be broken and crashes can occur. A best practice is to use raw interface pointers only to operate on interfaces temporarily, and not store them for later use.

GUIDs

You must use globally unique identifiers (GUIDs) to uniquely identify extensions and their interfaces. GUIDs are essentially 128-bit numbers generated by an algorithm to ensure they only exist once within a system such as Lumberyard. The use of GUIDs is key to implementing the type-safe casting of extension interfaces, which is particularly important in large scale development projects. To create GUIDs, you can use readily available tools like the **Create GUID** feature in Visual Studio or the macro below.

GUIDs are defined as follows.

```
struct CryGUID
{
 uint64 hipart;
 uint64 lopart;

 ...
};

typedef CryGUID CryInterfaceID;
typedef CryGUID CryClassID;
```

Declared in the following framework header files:

- `CryCommon/CryExtension/CryGUID.h`
- `CryCommon/CryExtension/CryTypeID.h`

The following Visual Studio macro can be used to generate GUIDs conveniently within the IDE. The macro writes GUIDs to the current cursor location in the source code editor window. Once added to **Macro Explorer**, the macro can be bound to a keyboard shortcut or (custom) toolbar.

```
Public Module CryGUIDGenModule

    Sub GenerateCryGUID()
        Dim newGuid As System.Guid
        newGuid = System.Guid.NewGuid()
        Dim guidStr As String

        guidStr = newGuid.ToString("N")
        guidStr = guidStr.Insert(16, ", 0x")
        guidStr = guidStr.Insert(0, "0x")

        REM guidStr = guidStr + vbNewLine
        REM guidStr = guidStr + newGuid.ToString("D")

        DTE.ActiveDocument.Selection.Text = guidStr
    End Sub

End Module
```

ICryUnknown

`ICryUnknown` provides the base interface for all extensions. If making it the top of the class hierarchy is not possible or desired (for example, in third party code), you can apply an additional level of indirection to expose the code by using the extension framework. For an example, see If ICryUnknown Cannot Be the Base of the Extension Class (p. 235).

ICryUnknown is declared as follows.

```
struct ICryUnknown
{
 CRYINTERFACE_DECLARE(ICryUnknown, 0x1000000010001000, 0x1000100000000000)

 virtual ICryFactory* GetFactory() const = 0;

protected:
 virtual void* QueryInterface(const CryInterfaceID& iid) const = 0;
 virtual void* QueryComposite(const char* name) const = 0;
};

typedef boost::shared_ptr<ICryUnknown> ICryUnknownPtr;
```

- `GetFactory()` returns the factory with which the specified extension object was instantiated. Using the provided glue code this function has constant runtime.

- `QueryInterface()` returns a void pointer to the requested interface if the extension implements it, or NULL otherwise. This function was deliberately declared as protected to enforce usage of type-safe interface casting semantics. For information on casting semantics, see Interface casting semantics (p. 216). When the provided glue code is used, this function has a (worst case) run time that is linear in the number of supported interfaces. Due to glue code implementation details, no additional internal function calls are needed. A generic code generator produces a series of instructions that compares interface IDs and returns a properly cast pointer.

- `QueryComposite()` returns a void pointer to the queried composite if the extension exposes it; otherwise, NULL. As with `QueryInterface()`, this function was deliberately declared as protected to enforce type querying. For information on type querying, see Querying composites (p. 217). The function has a (worst case) run time linear in the number of exposed composites.

- Unlike in COM, `ICryUnknown` does not have `AddRef()` and `Release()`. Reference counting is implemented in an non-intrusive way by using shared pointers that are returned by the framework when extension classes are instantiated.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

ICryFactory

`ICryFactory` provides the base interface to instantiate extensions. It is declared as follows.

```
struct ICryFactory
{
 virtual const char* GetClassName() const = 0;
 virtual const CryClassID& GetClassID() const = 0;
 virtual bool ClassSupports(const CryInterfaceID& iid) const = 0;
 virtual void ClassSupports(const CryInterfaceID*& pIIDs, size_t& numIIDs) const
 = 0;
 virtual ICryUnknownPtr CreateClassInstance() const = 0;

protected:
 virtual ~ICryFactory() {}
};
```

- `GetClassName()` returns the name of the extension class. This function has constant run time when the provided glue code is used.

- `GetClassID()` returns the ID of the extension class. This function has constant run time when the provided glue code is used.

- `ClassSupports(iid)` returns true if the interface with the specified ID is supported by the extension class; otherwise, false. This function has a (worst case) run time linear in the number of supported interfaces when the provided glue code is used.

- `ClassSupports(pIIDs, numIIDs)` returns the pointer to an internal array of IDs enumerating all of the interfaces that this extension class supports as well as the length of the array. This function has constant run time when the provided glue code is used.

- `CreateClassInstance()` dynamically creates an instance of the extension class and returns a shared pointer to it. If the extension class is implemented as a singleton, it will return a (static) shared pointer that wraps the single instance of that extension class. This function has constant run time when the provided glue code is used, except for the cost of the constructor call for non-singleton extensions.

- The destructor is declared protected to prevent explicit destruction from the client side by using `delete`, `boost::shared_ptr<T>`, etc. `ICryFactory` instances exist (as singletons) throughout the entire lifetime of any Lumberyard process and **must not** be destroyed.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryFactory.h`

ICryFactoryRegistry

ICryFactoryRegistry is a system-implemented interface that enables clients to query extensions. It is declared as follows.

```
struct ICryFactoryRegistry
{
 virtual ICryFactory* GetFactory(const char* cname) const = 0;
 virtual ICryFactory* GetFactory(const CryClassID& cid) const = 0;
 virtual void IterateFactories(const CryInterfaceID& iid, ICryFactory**
pFactories, size_t& numFactories) const = 0;

protected:
 virtual ~ICryFactoryRegistry() {}
};
```

- `GetFactory(cname)` returns the factory of the extension class with the specified name; otherwise, NULL.
- `GetFactory(cid)` returns the factory of the extension class with the specified ID; otherwise, NULL.
- `IterateFactory()` if `pFactories` is not NULL, `IterateFactory` copies up to `numFactories` entries of pointers to extension factories that support `iid`. `numFactories` returns the number of pointers copied. If `pFactories` is NULL, `numFactories` returns the total amount of extension factories that support `iid`.
- The destructor was declared protected to prevent explicit destruction from the client side by using `delete`, `boost::shared_ptr<T>`, etc. `ICryFactoryRegistry` is a system interface and that exists throughout the entire lifetime of any CryEngine process and **must not** be destroyed.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryFactoryRegistry.h`

Additional Extensions

Use the methods defined in `ICryUnknown` for additional functionality.

Interface casting semantics

Interface casting semantics have been implemented to provide syntactically convenient and type-safe casting of interfaces. The syntax was designed to conform with traditional C++ type casts and respects `const` rules.

```
ICryFactory* pFactory = ...;
assert(pFactory);

ICryUnknownPtr pUnk = pFactory->CreateClassInstance();

IMyExtensionPtr pMyExtension = cryinterface_cast<IMyExtension>(pUnk);
```

```
if (pMyExtension)
{
 // it's safe to work with pMyExtension
}
```

Interface casting also works on raw interface pointers. However, please consider the guidelines described in the section Shared and raw interface pointers (p. 213).

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Querying interface identifiers

Occasionally, it is necessary to know the ID of an interface, e.g. to pass it to `ICryFactoryRegistry::IterateFactories()`. This can be done as follows.

```
CryInterfaceID iid = cryiidof<IMyExtension>();
```

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Checking pointers

Use this extension to check whether pointers to different interfaces belong to the same class instance.

```
IMyExtensionAPtr pA = ...;
IMyExtensionBPtr pB = ...;

if (CryIsSameClassInstance(pA, pB))
{
 ...
}
```

This works on both shared and raw interface pointers.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Querying composites

Extensions can be queried for composites as follows.

```
IMyExtensionPtr pMyExtension = ...;

ICryUnknownPtr pCompUnk = crycomposite_query(pMyExtension, "foo");

IFooPtr pComposite = cryinterface_cast<IFoo>(pCompUnk);
if (pComposite)
{
 // it's safe to work with pComposite, a composite of pMyExtention exposed as
"foo" implementing IFoo
}
```

A call to `crycomposite_query()` might return NULL if the specified composite has not yet been created. To gather more information, the query can be rewritten as follows.

```
IMyExtensionPtr pMyExtension = ...;

bool exposed = false;
ICryUnknownPtr pCompUnk = crycomposite_query(pMyExtension, "foo", &exposed);

if (exposed)
{
 if (pCompUnk)
 {
  // "foo" exposed and created

  IFooPtr pComposite = cryinterface_cast<IFoo>(pCompUnk);
  if (pComposite)
  {
   // it's safe to work with pComposite, a composite of pMyExtention exposed
as "foo" implementing IFoo
  }
 }
 else
 {
  // "foo" exposed but not yet created
 }
}
else
{
 // "foo" not exposed by pMyExtension
}
```

As with interface casting composite, queries work on raw interface pointers. However, please consider the guidelines described in the section Shared and raw interface pointers (p. 213).

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Glue Code Macros

The following macros provide glue code to implement the base interfaces and services to support the framework in a thread-safe manner. You are strongly encouraged to use them when you implement an extension.

For examples of how these macros work together, see Using Glue Code (p. 227).

Declared in the following framework header files:

- `CryCommon/CryExtension/Impl/ClassWeaver.h`
- `CryCommon/CryExtension/CryGUID.h`

CRYINTERFACE_DECLARE(iname, iidHigh, iidLow)

Declares an interface and associated ID. Protects the interfaces from accidentally being deleted on client side. That is, it allows destruction only by using `boost::shared_ptr<T>`. This macro is required once per interface declaration.

Parameters

iname
The (C++) name of the interface as declared.

iidHigh
The higher 64-bit part of the interface ID (GUID).

iidLow
The lower 64-bit part of the interface ID (GUID).

CRYINTERFACE_BEGIN()

Start marker of the interface list inside the extension class implementation. Required once per extension class declaration.

CRYINTERFACE_ADD(iname)

Marker to add interfaces inside the extension class declaration. It has to be declared in between `CRYINTERFACE_BEGIN()` and any of the `CRYINTERFACE_END*()` markers. Only declare the interfaces that the class directly inherits. If deriving from an existing extension class or classes, the inherited interfaces get added automatically. If an interface is declared multiple times, duplicates will be removed. It is not necessary to add `ICryUnknown`.

Caution
Other interfaces that are not declared will not be castable by using `cryinterface_cast<T>()`.

Parameters

iname
The (C++) name of the interface to be added.

CRYINTERFACE_END()

End marker of the interface list inside the extension class declaration. Use this if not inheriting from any already existing extension class. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

CRYINTERFACE_ENDWITHBASE(base)

End marker of the interface list inside the extension class declaration. Use this if inheriting from an already existing extension class. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

Parameters

base
The (C++) name of the extension class from which derived.

CRYINTERFACE_ENDWITHBASE2(base0, base1)

End marker of the interface list inside the extension class declaration. Use this if inheriting from two already existing extension classes. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

Parameters

base0
The (C++) name of the first extension class from which derived.

base1
The (C++) name of the second extension class from which derived.

CRYINTERFACE_ENDWITHBASE3(base0, base1, base2)

End marker of the interface list inside the extension class declaration. Use this if inheriting from three already existing extension classes. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

Parameters

base0
The (C++) name of the first extension class from which derived.

base1
The (C++) name of the second extension class from which derived.

base2
The (C++) name of the 3rd extension class from which derived.

CRYINTERFACE_SIMPLE(iname)

Convenience macro for the following code sequence (probably the most common extension case):

```
CRYINTERFACE_BEGIN()
 CRYINTERFACE_ADD(iname)
CRYINTERFACE_END()
```

Parameters

iname
The (C++) name of the interface to be added.

CRYCOMPOSITE_BEGIN()

Start marker of the list of exposed composites.

CRYCOMPOSITE_ADD(member, membername)

Marker to add a member of the extension class to the list of exposed composites.

Parameters

member
The (C++) name of the extension class member variable to be exposed. It has to be of type `boost::shared_ptr<T>`, where `T` inherits from `ICryUnknown`. This condition is enforced at compile time.

membername
The name (as C-style string) of the composite by which the composite can later be queried at runtime.

CRYCOMPOSITE_END(implclassname)

End marker of the list of exposed composites. Use this if not inheriting from any extension class that also exposes composites. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname
The (C++) name of the extension class to be implemented.

CRYCOMPOSITE_ENDWITHBASE(implclassname, base)

End marker of the list of exposed composites. Use this if inheriting from one extension class that also exposes composites. Queries will first search in the current class and then look into the base class to find a composite that matches the requested name specified in `crycomposite_query()`. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname
The (C++) name of the extension class to be implemented.

base
The (C++) name of the extension class derived from.

CRYCOMPOSITE_ENDWITHBASE2(implclassname, base0, base1)

End marker of the list of exposed composites. Use this if inheriting from two extension classes that also expose composites. Queries will first search in the current class and then look into the base classes to find a composite matching the requested name specified in `crycomposite_query()`. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname
The (C++) name of the extension class to be implemented.

base0
The (C++) name of the first extension class from which derived.

base1
The (C++) name of the second extension class which derived.

CRYCOMPOSITE_ENDWITHBASE3(implclassname, base0, base1, base2)

End marker of the list of exposed composites. Use this if inheriting from three extension classes that also expose composites. Queries will first search in the current class and then look into the base classes to find a composite matching the requested name specified in `crycomposite_query()`. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname
The (C++) name of the extension class to be implemented.

base0
The (C++) name of the first extension class from which derived.

base1
The (C++) name of the second extension class from which derived.

base2
The (C++) name of the third extension class from which derived.

CRYGENERATE_CLASS(implclassname, cname, cidHigh, cidLow)

Generates code to support base interfaces and services for an extension class that can be instantiated an arbitrary number of times. Required once per extension class declaration. Mutually exclusive to CRYGENERATE_SINGLETONCLASS().

Parameters

implclassname
The C++ class name of the extension.

cname
The extension class name with which it is registered in the registry.

cidHigh
The higher 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

cidLow
The lower 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

CRYGENERATE_SINGLETONCLASS(implclassname, cname, cidHigh, cidLow)

Generates code to support base interfaces and services for an extension class that can be instantiated only once (singleton). Required once per extension class declaration. Mutually exclusive with `CRYGENERATE_CLASS()`.

Parameters

implclassname
The C++ class name of the extension.

cname
The extension class name with which it is registered in the registry.

cidHigh
The higher 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

cidLow
The lower 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

CRYREGISTER_CLASS(implclassname)

Registers the extension class in the system. Required once per extension class at file scope.

Parameters

implclassname
The C++ class name of the extension.

MAKE_CRYGUID(high, low)

Parameters

Constructs a CryGUID. Useful when searching the registry for extensions by class ID.

high
The higher 64-bit part of the GUID.

low
The lower 64-bit part of the GUID.

CryExtension Samples

Sample 1 - Implementing a Source Control Plugin by Using Extensions

```
//////////////////////////////////////////////////////////////////////////
// source control interface

struct ISourceControl : public ICryUnknown
{
 CRYINTERFACE_DECLARE(ISourceControl, 0x399d8fc1d94044cc, 0xa70d2b4e58921453)

 virtual void GetLatest(const char* filename) = 0;
 virtual void Submit() = 0;
};

typedef cryshared_ptr<ISourceControl> ISourceControlPtr;

//////////////////////////////////////////////////////////////////////////
// concrete implementations of source control interface

class CSourceControl_Perforce : public ISourceControl
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(ISourceControl)
 CRYINTERFACE_END()
 CRYGENERATE_SINGLETONCLASS(CSourceControl_Perforce, "CSourceControl_Perforce",
 0x7305bff20ee543e3, 0x820792c56e74ecda)

 virtual void GetLatest(const char* filename) { ... };
 virtual void Submit() { ... };
};

CRYREGISTER_CLASS(CSourceControl_Perforce)


class CSourceControl_SourceSafe : public ISourceControl
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(ISourceControl)
 CRYINTERFACE_END()

 CRYGENERATE_SINGLETONCLASS(CSourceControl_SourceSafe, "CSourceCon
trol_SourceSafe", 0x1df62628db9d4bb2, 0x8164e418dd5b6691)

 virtual void GetLatest(const char* filename) { ... };
 virtual void Submit() { ... };
};

CRYREGISTER_CLASS(CSourceControl_SourceSafe)

//////////////////////////////////////////////////////////////////////////
// using the interface (submitting changes)

void Submit()
{
 ICryFactoryRegistry* pReg = gEnv->pSystem->GetFactoryRegistry();

 ICryFactory* pFactory = 0;
 size_t numFactories = 1;
 pReg->IterateFactories(cryiidof<ISourceControl>(), &pFactory, numFactories);

 if (pFactory)
 {
  ISourceControlPtr pSrcCtrl = cryinterface_cast<ISourceControl>(pFactory-
>CreateClassInstance());
  if (pSrcCtrl)
  {
   pSrcCtrl->Submit();
  }
 }
}
```

Using Extensions

Working with Specific Extension Classes

To work with a specific extension class, a client needs to know the extension's class name or class id and the interface(s) that the class supports. With this information, the class factory can be queried from the registry, an instance created and worked with as in the following example.

```
// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 ...
};

typedef boost::shared_ptr<IMyExtension> IMyExtensionPtr;
```

```
// in client code
#include <IMyExtension.h>
#include <CryExtension/CryCreateClassInstance.h>

IMyExtensionPtr pMyExtension;

#if 0
// create extension by class name
if (CryCreateClassInstance("MyExtension", pMyExtension))
#else
// create extension by class id, guaranteed to create instance of same kind
if (CryCreateClassInstance(MAKE_CRYGUID(0x68c7f0e0c36446fe, 0x82a3bc01b54dc7bf),
 pMyExtension))
#endif
{
 // it's safe to work with pMyExtension
}
```

```
// verbose version of client code above
#include <IMyExtension.h>
#include <CryExtension/ICryFactory.h>
#include <CryExtension/ICryFactoryRegistry.h>

ICryFactoryRegistry* pReg = ...;

#if 0
// search extension by class name
ICryFactory* pFactory = pReg->GetFactory("MyExtension");
#else
// search extension by class id, guaranteed to yield same factory as in search
 by class name
ICryFactory* pFactory = pReg->GetFactory(MAKE_CRYGUID(0x68c7f0e0c36446fe,
0x82a3bc01b54dc7bf));
#endif

if (pFactory) // see comment below
{
 ICryUnknownPtr pUnk = pFactory->CreateClassInstance();
 IMyExtensionPtr pMyExtension = cryinterface_cast<IMyExtension>(pUnk);
 if (pMyExtension)
 {
  // it's safe to work with pMyExtension
 }
}
```

As an optimization, you can enhance the `if` check as follows.

```
if (pFactory && pFactory->ClassSupports(cryiidof<IMyExtension>()))
{
 ...
```

This version of the `if` statement will check interface support before the extension class is instantiated. This check prevents the unnecessary (and potentially expensive) construction and destruction of extensions that are incompatible with a given interface.

Finding Extension Classes that Support a Specific Interface

To determine how many extension classes in the registry support a given interface, and to list them, clients can submit queries similar to the following.

```
// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 ...
};

// in client code
#include <IMyExtension.h>
#include <CryExtension/ICryFactory.h>
#include <CryExtension/ICryFactoryRegistry.h>

ICryFactoryRegistry* pReg = ...;

size_t numFactories = 0;
pReg->IterateFactories(cryiidof<IMyExtension>(), 0, numFactories);

ICryFactory** pFactories = new ICryFactory*[numFactories];

pReg->IterateFactories(cryiidof<IMyExtension>(), pFactories, numFactories);

...

delete [] pFactories;
```

Implementing Extensions Using the Framework

The following section explains in detail how to implement extensions in Lumberyard. It provides examples that use glue code and do not use glue code. The section also shows you how to utilize the framework in cases where `ICryUnknown` cannot be the base of the extension interface.

Recommended Layout for Including Framework Header Files

The public interface header that will be included by the client should look like the following.

```
// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 ...
};
```

If you are using glue code, declare the implementation class of the extension in the header file as follows.

```
// MyExtension.h
#include <IMyExtension.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtension : public IMyExtension
{
 ...
};
```

Using Glue Code

The first example shows a possible implementation of the `IMyExtension` class in the previous examples.

```
///////////////////////////////////////////
// public section

// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IMyExtension, 0x4fb87a5f83f74323, 0xa7e42ca947c549d8)

 virtual void CallMe() = 0;
};

typedef boost::shared_ptr<IMyExtension> IMyExtensionPtr;

///////////////////////////////////////////
// private section not visible to client

// MyExtension.h
#include <IMyExtension.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtension : public IMyExtension
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(IMyExtension)
 CRYINTERFACE_END()

 CRYGENERATE_CLASS(CMyExtension, "MyExtension", 0x68c7f0e0c36446fe,
0x82a3bc01b54dc7bf)

public:
 virtual void CallMe();
};

// MyExtension.cpp
#include "MyExtension.h"
```

```
CRYREGISTER_CLASS(CMyExtension)

CMyExtension::CMyExtension()
{
}

CMyExtension::~CMyExtension()
{
}

void CMyExtension::CallMe()
{
 printf("Inside CMyExtension::CallMe()...");
}
```

The following example shows how the extension class `MyExtension` can be customized and expanded to implement two more interfaces, `IFoo` and `IBar`.

```
//////////////////////////////////////////
// public section

// IFoo.h
#include <CryExtension/ICryUnknown.h>

struct IFoo : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IFoo, 0x7f073239d1e6433f, 0xb59c1b6ff5f68d79)

 virtual void Foo() = 0;
};

// IBar.h
#include <CryExtension/ICryUnknown.h>

struct IBar : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IBar, 0xa9361937f60d4054, 0xb716cb711970b5d1)

 virtual void Bar() = 0;
};

//////////////////////////////////////////
// private section not visible to client

// MyExtensionCustomized.h
#include "MyExtension.h"
#include <IFoo.h>
#include <IBar.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtensionCustomized : public CMyExtension, public IFoo, public IBar
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(IFoo)
  CRYINTERFACE_ADD(IBar)
 CRYINTERFACE_ENDWITHBASE(CMyExtension)
 CRYGENERATE_CLASS(CMyExtensionCustomized, "MyExtensionCustomized",
0x07bfa7c543a64f0c, 0x861e9fa3f7d7d264)

public:
 virtual void CallMe(); // chose to override MyExtension's impl
 virtual void Foo();
 virtual void Bar();
};

// MyExtensionCustomized.cpp
#include "MyExtensionCustomized.h"

CRYREGISTER_CLASS(CMyExtensionCustomized)

CMyExtensionCustomized::CMyExtensionCustomized()
{
}

CMyExtensionCustomized::~CMyExtensionCustomized()
{
}

void CMyExtensionCustomized::CallMe()
{
 printf("Inside CMyExtensionCustomized::CallMe()...");
}

void CMyExtensionCustomized::Foo()
{
 printf("Inside CMyExtensionCustomized::Foo()...");
}

void CMyExtensionCustomized::Bar()
{
 printf("Inside CMyExtensionCustomized::Bar()...");
}
```

Without Using Glue Code

If for any reason using the glue code is neither desired nor applicable, extensions can be implemented as follows. It is recommended to implement `ICryUnknown` and `ICryFactory` such that their runtime cost is equal to the one provided by the glue code. For more information, see ICryUnknown (p. 214) and ICryFactory (p. 215).

```
////////////////////////////////////////////
// public section

// INoMacros.h
#include <CryExtension/ICryUnknown.h>

struct INoMacros : public ICryUnknown
{
 // befriend cryiidof and boost::checked_delete
 template <class T> friend const CryInterfaceID& InterfaceCastSe
mantics::cryiidof();
```

```
 template <class T> friend void boost::checked_delete(T* x);
protected:
 virtual ~INoMacros() {}

private:
 // It's very important that this static function is implemented for each inter
face!
 // Otherwise the consistency of cryinterface_cast<T>() is compromised because

 // cryiidof<T>() = cryiidof<baseof<T>>() {baseof<T> = ICryUnknown in most
cases}
 static const CryInterfaceID& IID()
 {
  static const CryInterfaceID iid = {0xd0fda1427dee4cceull,
0x88ff91b6b7be2a1full};
  return iid;
 }

public:
 virtual void TellMeWhyIDontLikeMacros() = 0;
};

typedef boost::shared_ptr<INoMacros> INoMacrosPtr;

///////////////////////////////////////////
// private section not visible to client

// NoMacros.cpp
//
// This is just an exemplary implementation!
// For brevity the whole implementation is packed into this cpp file.

#include <INoMacros.h>
#include <CryExtension/ICryFactory.h>
#include <CryExtension/Impl/RegFactoryNode.h>

// implement factory first
class CNoMacrosFactory : public ICryFactory
{
 // ICryFactory
public:
 virtual const char* GetClassName() const
 {
  return "NoMacros";
 }
 virtual const CryClassID& GetClassID() const
 {
  static const CryClassID cid = {0xa4550317690145c1ull, 0xa7eb5d85403dfad4ull};

  return cid;
 }
 virtual bool ClassSupports(const CryInterfaceID& iid) const
 {
  return iid == cryiidof<ICryUnknown>() || iid == cryiidof<INoMacros>();
 }
 virtual void ClassSupports(const CryInterfaceID*& pIIDs, size_t& numIIDs) const

 {
  static const CryInterfaceID iids[2] = {cryiidof<ICryUnknown>(), cryiidof<IN
oMacros>()};
  pIIDs = iids;
  numIIDs = 2;
 }
 virtual ICryUnknownPtr CreateClassInstance() const;

public:
 static CNoMacrosFactory& Access()
 {
  return s_factory;
 }

private:
 CNoMacrosFactory() {}
 ~CNoMacrosFactory() {}

private:
 static CNoMacrosFactory s_factory;
};

CNoMacrosFactory CNoMacrosFactory::s_factory;

// implement extension class
class CNoMacros : public INoMacros
{
 // ICryUnknown
public:
 virtual ICryFactory* GetFactory() const
 {
  return &CNoMacrosFactory::Access();
 };

 // befriend boost::checked_delete
 // only needed to be able to create initial shared_ptr<CNoMacros>
 // so we don't lose type info for debugging (i.e. inspecting shared_ptr)
 template <class T> friend void boost::checked_delete(T* x);

protected:
 virtual void* QueryInterface(const CryInterfaceID& iid) const
 {
  if (iid == cryiidof<ICryUnknown>())
   return (void*) (ICryUnknown*) this;
  else if (iid == cryiidof<INoMacros>())
   return (void*) (INoMacros*) this;
  else
   return 0;
 }

 virtual void* QueryComposite(const char* name) const
 {
  return 0;
 }

 // INoMacros
public:
 virtual void TellMeWhyIDontLikeMacros()
 {
  printf("Woohoo, no macros...\n");
 }

 CNoMacros() {}

protected:
 virtual ~CNoMacros() {}
};

// implement factory's CreateClassInstance method now that extension class is
fully visible to compiler
ICryUnknownPtr CNoMacrosFactory::CreateClassInstance() const
{
 boost::shared_ptr<CNoMacros> p(new CDontLikeMacros);
 return ICryUnknownPtr(*static_cast<boost::shared_ptr<ICryUnknown>*>(stat
ic_cast<void*>(&p)));
}

// register extension
static SRegFactoryNode g_noMacrosFactory(&CNoMacrosFactory::Access());
```

Exposing Composites

The following example shows how to expose (inherited) composites. For brevity, the sample is not separated into files.

```
 //////////////////////////////////////////////////////////////////////////
 struct ITestExt1 : public ICryUnknown
 {
  CRYINTERFACE_DECLARE(ITestExt1, 0x9d9e0dcfa5764cb0, 0xa73701595f75bd32)

  virtual void Call1() = 0;
 };

 typedef boost::shared_ptr<ITestExt1> ITestExt1Ptr;

 class CTestExt1 : public ITestExt1
 {
  CRYINTERFACE_BEGIN()
   CRYINTERFACE_ADD(ITestExt1)
  CRYINTERFACE_END()

  CRYGENERATE_CLASS(CTestExt1, "TestExt1", 0x43b04e7cc1be45ca,
0x9df6ccb1c0dc1ad8)

 public:
  virtual void Call1();
 };

 CRYREGISTER_CLASS(CTestExt1)

 CTestExt1::CTestExt1()
 {
 }

 CTestExt1::~CTestExt1()
 {
 }

 void CTestExt1::Call1()
 {
 }

 ///////////////////////////////////////////////////////////////////////////
 class CComposed : public ICryUnknown
 {
  CRYINTERFACE_BEGIN()
  CRYINTERFACE_END()

  CRYCOMPOSITE_BEGIN()
   CRYCOMPOSITE_ADD(m_pTestExt1, "Ext1")
  CRYCOMPOSITE_END(CComposed)

  CRYGENERATE_CLASS(CComposed, "Composed", 0x0439d74b8dcd4b7f,
0x9287dcdf7e26a3a5)

 private:
  ITestExt1Ptr m_pTestExt1;
 };

 CRYREGISTER_CLASS(CComposed)

 CComposed::CComposed()
 : m_pTestExt1()
 {
  CryCreateClassInstance("TestExt1", m_pTestExt1);
 }

 CComposed::~CComposed()
 {
 }

 ///////////////////////////////////////////////////////////////////////////
 struct ITestExt2 : public ICryUnknown
 {
  CRYINTERFACE_DECLARE(ITestExt2, 0x8eb7a4b399874b9c, 0xb96bd6da7a8c72f9)

  virtual void Call2() = 0;
 };

 DECLARE_BOOST_POINTERS(ITestExt2);

 class CTestExt2 : public ITestExt2
 {
  CRYINTERFACE_BEGIN()
   CRYINTERFACE_ADD(ITestExt2)
  CRYINTERFACE_END()

  CRYGENERATE_CLASS(CTestExt2, "TestExt2", 0x25b3ebf8f1754b9a,
0xb5494e3da7cdd80f)

 public:
  virtual void Call2();
 };
```

```
CRYREGISTER_CLASS(CTestExt2)

CTestExt2::CTestExt2()
{
}

CTestExt2::~CTestExt2()
{
}

void CTestExt2::Call2()
{
}

//////////////////////////////////////////////////////////////////////////
class CMultiComposed : public CComposed
{
 CRYCOMPOSITE_BEGIN()
  CRYCOMPOSITE_ADD(m_pTestExt2, "Ext2")
 CRYCOMPOSITE_ENDWITHBASE(CMultiComposed, CComposed)

 CRYGENERATE_CLASS(CMultiComposed, "MultiComposed", 0x0419d74b8dcd4b7e,
0x9287dcdf7e26a3a6)

private:
 ITestExt2Ptr m_pTestExt2;
};

CRYREGISTER_CLASS(CMultiComposed)

CMultiComposed::CMultiComposed()
: m_pTestExt2()
{
 CryCreateClassInstance("TestExt2", m_pTestExt2);
}

CMultiComposed::~CMultiComposed()
{
}

...

//////////////////////////////////////////////////////////////////////////
// let's use it

ICryUnknownPtr p;
if (CryCreateClassInstance("MultiComposed", p))
{
 ITestExt1Ptr p1 = cryinterface_cast<ITestExt1>(crycomposite_query(p, "Ext1"));

 if (p1)
  p1->Call1(); // calls CTestExt1::Call1()
 ITestExt2Ptr p2 = cryinterface_cast<ITestExt2>(crycomposite_query(p, "Ext2"));

 if (p2)
  p2->Call2(); // calls CTestExt2::Call2()
 }
```

If ICryUnknown Cannot Be the Base of the Extension Class

There are cases where making `ICryUnknown` the base of your extension class is not possible. Some examples are legacy code bases that cannot be modified, third party code for which you do not have full source code access, or code whose modification is not practical. However, these code bases can provide useful functionality (for example, for video playback or flash playback) if you expose them as engine extensions. The following sample illustrates how an additional level of indirection can expose a third party API.

```
//////////////////////////////////////////////
// public section

// IExposeThirdPartyAPI.h
#include <CryExtension/ICryUnknown.h>
#include <IThirdPartyAPI.h>

struct IExposeThirdPartyAPI : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IExposeThirdPartyAPI, 0x804250bbaacf4a5f,
0x90ef0327bb7a0a7f)

 virtual IThirdPartyAPI* Create() = 0;
};

typedef boost::shared_ptr<IExposeThirdPartyAPI> IExposeThirdPartyAPIPtr;

//////////////////////////////////////////////
// private section not visible to client

// Expose3rdPartyAPI.h
#include <IExposeThirdPartyAPI.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CExposeThirdPartyAPI : public IExposeThirdPartyAPI
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(IExposeThirdPartyAPI)
 CRYINTERFACE_END()

 CRYGENERATE_CLASS(CExposeThirdPartyAPI, "ExposeThirdPartyAPI",
0xa93b970b2c434a21, 0x86acfe94d8dae547)

public:
 virtual IThirdPartyAPI* Create();
};

// ExposeThirdPartyAPI.cpp
#include "ExposeThirdPartyAPI.h"
#include "ThirdPartyAPI.h"

CRYREGISTER_CLASS(CExposeThirdPartyAPI)

CExposeThirdPartyAPI::CExposeThirdPartyAPI()
```

```
{
}

CExposeThirdPartyAPI::~CExposeThirdPartyAPI()
{
}

IThirdPartyAPI* CExposeThirdPartyAPI::Create()
{
 return new CThirdPartyAPI; // CThirdPartyAPI implements IThirdPartyAPI
}
```

Custom Inclusion and Exclusion of Extensions

To enable easy inclusion and exclusion of extensions, Lumberyard provides a global "extension definition" header much like `CryCommon/ProjectDefines.h` that is automatically included in all modules by means of the `platform.h` file. To wrap your extension implementation code, you include a `#define` statement in the extension definition header. To exclude unused extension code from your build, you can also comment out extensions that you are not interested in. Interface headers are not affected by the `#if defined` statements, so the client code compiles as is with or without them.

```
///////////////////////////////////////////
// public section

// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 ...
};

typedef boost::shared_ptr<IMyExtension> IMyExtensionPtr;

// ExtensionDefines.h
...
#define INCLUDE_MYEXTENSION
...

///////////////////////////////////////////
// private section not visible to client

// MyExtension.h
#if defined(INCLUDE_MYEXTENSION)

#include <IMyExtension.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtension : public IMyExtension
{
 ...
};

#endif // #if defined(INCLUDE_MYEXTENSION)

// MyExtension.cpp
```

```
#if defined(INCLUDE_MYEXTENSION)

#include "MyExtension.h"

CRYREGISTER_CLASS(CMyExtension)

...

#endif // #if defined(INCLUDE_MYEXTENSION)
```

Because extensions can be removed from a build, clients must write their code in a way that does not assume the availability of an extension. For more information, see Using Extensions (p. 224).

CryString

Lumberyard has a custom reference-counted string class `CryString` (declared in `CryString.h`) which is a replacement for STL `std::string`. `CryString` should always be preferred over `std::string`. For convenience, `string` is used as a typedef for `CryString`.

How to Use Strings as Key Values for STL Containers

The following code shows good (efficient) and bad usage:

```
const char *szKey= "Test";

map< string, int >::const_iterator iter = m_values.find( CONST_TEMP_STRING(
szKey ) );   // Good

map< string, int >::const_iterator iter = m_values.find( szKey );  // Bad
```

By using the suggested method, you avoid the allocation, deallocation, and copying of a temporary string object, which is a common problem for most string classes. By using the macro `CONST_TEMP_STRING`, the string class uses the pointer directly without having to free data afterwards.

Further Usage Tips

- Do not use `std::string` or `std::wstring`. Instead, use only `string` and `wstring`, and never include the standard string header `<string>`.
- Use the `c_str()` method to access the contents of the string.
- Because strings are reference-counted, never modify memory returned by the `c_str()` method. Doing so could affect the wrong string instance.
- Do not pass strings via abstract interfaces; all interfaces should use `const char*` in interface methods.
- `CryString` has a combined interface of `std::string` and the MFC `CString`, so you can use both interface types for string operations.
- Avoid doing many string operations at runtime as they often cause memory reallocations.
- For fixed size strings (e.g. 256 chars), use `CryFixedStringT`, which should be preferred over static `char` arrays.

ICrySizer

The `ICrySizer` interface can be implemented to record detailed information about the memory usage of a class.

Note
This information is also available in the Editor under **Engine Memory info**.

How to use the ICrySizer interface

The following example shows how to use the `ICrySizer` interface.

```
void GetMemoryUsage( ICrySizer *pSizer )
{
 {
    SIZER_COMPONENT_NAME( pSizer, "Renderer (Aux Geometries)" );
    pSizer->Add(*this);
 }
 pSizer->AddObject(<element_prow>,<element_count>);
 pSizer->AddObject(<container>);
 m_SubObject.GetMemoryUsage(pSizer);
}
```

Serialization Library

The `CryCommon` serialization library has the following features:

- Separation of user serialization code from the actual storage format. This makes it possible to switch between XML, JSON, and binary formats without changing user code.
- Re-usage of the same serialization code for editing in the PropertyTree. You can write the serialization code once and use it to expose your structure in the editor as a parameters tree.
- Enables you to write serialization code in non-intrusive way (as global overloaded functions) without modifying serialized types.
- Makes it easy to change formats. For example, you can add, remove, or rename fields and still be able to load existing data.

Tutorial

The example starts with a data layout that uses standard types, enumerations, and containers. The example adds the `Serialize` method to structures with fixed signatures.

Defining data

```
#include "Serialization/IArchive.h"
#include "Serialization/STL.h"

enum AttachmentType
{
 ATTACHMENT_SKIN,
 ATTACHMENT_BONE
};
struct Attachment
{
 string name;
 AttachmentType type;
 string model;
 void Serialize(Serialization::IArchive& ar)
 {
  ar(name, "name", "Name");
  ar(type, "type", "Type");
  ar(model, "model", "Model");
 }
};
struct Actor
{
 string character;
 float speed;
 bool alive;
 std::vector<Attachment> attachments;
 Actor()
 : speed(1.0f)
 , alive(true)
 {
 }
 void Serialize(Serialization::IArchive& ar)
 {
  ar(character, "character", "Character");
  ar(speed, "speed", "Speed");
  ar(alive, "alive", "Alive");
  ar(attachments, "attachments", "Attachment");
 }
};

// Implementation file:
#include "Serialization/Enum.h"

SERIALIZATION_ENUM_BEGIN(AttachmentType, "Attachment Type")
SERIALIZATION_ENUM(ATTACHMENT_BONE, "bone", "Bone")
SERIALIZATION_ENUM(ATTACHMENT_SKIN, "skin", "Skin")
SERIALIZATION_ENUM_END()
```

Why are two names needed?

The `ar()` call takes two string arguments: one is called `name`, and the second `label`. The `name` argument is used to store parameters persistently; for example, for JSON and XML. The `label` parameter is used for the PropertyTree. The `label` parameter is typically longer, more descriptive, contains white space, and may be easily changed without breaking compatibility with existing data. In contrast, `name` is a C-style identifier. It is also convenient to have `name` match the variable name so that developers can easily find the variable by looking at the data file.

Omitting the `label` parameter (the equivalent of passing `nullptr`) will hide the parameter in the PropertyTree, but it will be still serialized and can be copied together with its parent by using copy-paste.

Note

The `SERIALIZATION_ENUM` macros should reside in the .cpp implementation file because they contain symbol definitions.

Serializing into or from a file

Now that the data has been defined, it is ready for serialization. To implement the serialization, you can use `Serialization::SaveJsonFile`, as in the following example.

```
#include <Serialization/IArchiveHost.h>


Actor actor;
Serialization::SaveJsonFile("filename.json", actor);
```

This will output content in the following format:

```
{
 "character": "nanosuit.cdf",
 "speed": 2.5,
 "alive": true,
 "attachments": [
  { "name": "attachment 1", "type": "bone", "model": "model1.cgf" },
  { "name": "attachment 2", "type": "skin", "model": "model2.cgf" }
 ]
}
```

The code for reading data is similar to that for serialization, except that it uses `Serialization::LoadJsonFile`.

```
#include <Serialization/IArchiveHost.h>

Actor actor;
Serialization::LoadJsonFile(actor, "filename.json");
```

The save and load functions used are wrappers around the `IArchiveHost` interface, an instance of which is located in `gEnv->pSystem->GetArchiveHost()`. However, if you have direct access to the archive code (for example, in `CrySystem` or `EditorCommon`), you can use the archive classes directly, as in the following example.

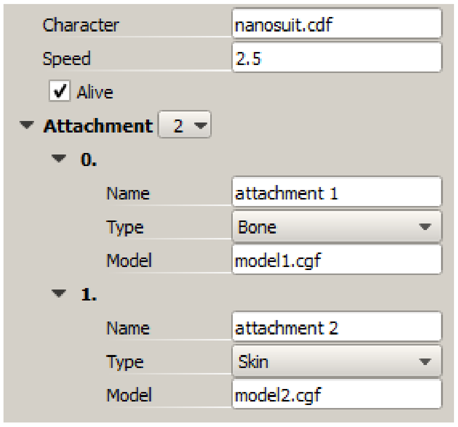
```
#include <Serialization/JSONOArchive.h>
#include <Serialization/JSONIArchive.h>

Serialization::JSONOArchive oa;

Actor actor;
oa(actor);
oa.save("filename.json");

// to get access to the data without saving:
const char* jsonString = oa.c_str();

// and to load
Serialization::JSONIArchive ia;
if (ia.load("filename.json"))
{
 Actor loadedActor;
 ia(loadedActor);
}
```

Editing in the PropertyTree

If you have the `Serialize` method implemented for your types, it is easy to get it exposed to the `QPropertyTree`, as the following example shows.

```
#include <QPropertyTree/QPropertyTree.h>

QPropertyTree* tree = new QPropertyTree(parent);

static Actor actor;
tree->attach(Serialization::SStruct(actor));
```

You can select enumeration values from the list and add or remove vector elements by using the [2] button or the context menu.



In the moment of attachment, the `Serialize` method will be called to extract properties from your object. As soon as the user changes a property in the UI, the `Serialize` method is called to write properties back to the object.

> **Note**
> It is important to remember that `QPropertyTree` holds a reference to an attached object. If the object's lifetime is shorter than the tree, an explicit call to `QPropertyTree::detach()` should be performed.

Use Cases

Non-intrusive serialization

Normally when `struct` or a class instance is passed to the archive, the `Serialize` method of the instance is called. However, it is possible to override this behavior by declaring the following global function:

```
bool Serialize(Serialization::IArchive&, Type& value, const char* name, const
char* label);
```

The return value here has the same behavior as `IArchive::operator()`. For input archives, the function returns false when a field is missing or wasn't read. For output archives, it always returns true.

Note
The return value does not propagate up. If one of the nested fields is missing, the top level block will still return true.

The global function approach is useful when you want to:

- Add serialization in non-intrusive way
- Transform data during serialization
- Add support for unsupported types like plain pointers

The following example adds support for `std::pair<>` type to the `Serialize` function:

```
template<class T1, class T2>
struct pair_serializable : std::pair<T1, T2>
{
  void Serialize(Serialization::IArchive& ar)
  {
    ar(first, "first", "First");
    ar(second, "second", "Second");
  }
}

template<class T1, class T2>
bool Serialize(Serialization::IArchive& ar, std::pair<T1, T2>& value, const
char* name, const char* label)
{
  return ar(static_cast<pair_serializable<T1, T2>&>(value), name, label);
}
```

The benefit of using inheritance is that you can get access to protected fields. In cases when access policy is not important and inheritance is undesirable, you can replace the previous code with following pattern.

```
template<class T1, class T2>
struct pair_serializable
{
  std::pair<T1, T2>& instance;

  pair_serializable(std::pair<T1, T2>& instance) : instance(instance) {}
  void Serialize(Serialization::IArchive& ar)
  {
    ar(instance.first, "first", "First");
    ar(instance.second, "second", "Second");
  }
}

template<class T1, class T2>
bool Serialize(Serialization::IArchive& ar, std::pair<T1, T2>& value, const
char* name, const char* label)
{
  pair_serializable<T1, T2> serializer(value);
  return ar(serializer, name, label);
}
```

Registering Enum inside a Class

Normally, `SERIALIZATION_ENUM_BEGIN()` will not compile if you specify enumeration within a class (a "nested `enum`"). To overcome this shortcoming, use `SERIALIZATION_ENUM_BEGIN_NESTED`, as in the following example.

```
SERIALIZATION_ENUM_BEGIN_NESTED(Class, Enum, "Label")
SERIALIZATION_ENUM(Class::ENUM_VALUE1, "value1", "Value 1")
SERIALIZATION_ENUM(Class::ENUM_VALUE2, "value2", "Value 2")
SERIALIZATION_ENUM_END()
```

Polymorphic Types

The Serialization library supports the loading and saving of polymorphic types. This is implemented through serialization of a smart pointer to the base type.

For example, if you have following hierarchy:

IBase

- ImplementationA
- ImplementationB

You would need to register derived types with a macro, as in the following example.

```
SERIALIZATION_CLASS_NAME(IBase, ImplementationA, "impl_a", "Implementation A");
SERIALIZATION_CLASS_NAME(IBase, ImplementationA, "impl_b", "Implementation B");
```

Now you can serialize a pointer to the base type:

```
#include <Serialization/SmartPtr.h>

_smart_ptr<IInterfface> pointer;

ar(pointer, "pointer", "Pointer");
```

The first string is used to name the type for persistent storage, and the second string is a human-readable name for display in the PropertyTree.

Customizing presentation in the PropertyTree

There are two aspects that can be customized within the PropertyTree:

1. The layout of the property fields. These are controlled by control sequences in the label (the third argument in `IArchive::operator()`).
2. Decorators. These are defined in the same way that specific properties are edited or represented.

Control characters

Control sequences are added as a prefix to the third argument for `IArchive::operator()`. These characters control the layout of the property field in the PropertyTree.

Layout Control Characters

Prefix	Role	Description
!	Read-only field	Prevents the user from changing the value of the property. The effect is non-recursive.
^	Inline	Places the property on the same line as the name of the structure root. Can be used to put fields in one line in a horizontal layout, rather than in the default vertical list.
^^	Inline in front of a name	Places the property name before the name of the parent structure. Useful to add check boxes before a name.
<	Expand value field	Expand the value part of the property to occupy all available space.
>	Contract value field	Reduces the width of the value field to the minimum. Useful to restrict the width of inline fields.
>*N*>	Limit field width to *N* pixels	Useful for finer control over the UI. Not recommended for use outside of the editor.
+	Expand row by default.	Can be used to control which structures or containers are expanded by default. Use this only when you need per-item control. Otherwise, `QPropertyTree::setExpandLevels` is a better option.
[*S*]	Apply *S* control characters to children.	Applies control characters to child properties. Especially useful with containers.

Combining control characters

Multiple control characters can be put together to combine their effects, as in the following example.

```
ar(name, "name", "^!<Name"); // inline, read-only, expanded value field
```

Decorators

There are two kinds of decorators:

1. Wrappers that implement a custom serialization function that performs a transformation on the original value. For example, `Serialization/Math.h` contains `Serialization::RadiansAsDeg(float&)` that allows to store and edit angles in radians.

2. Wrappers that do no transformation but whose type is used to select a custom property implementation in the PropertyTree. Resource Selectors are examples of this kind of wrapper.

Decorator	Purpose	Defined for types	Context needed
`AnimationPath`	Selection UI for full animation path.	Any string-like type, like: `std::string`, `string (CryStringT)`, `SCRCRef` `CCryName`	
`CharacterPath`	UI: browse for character path (cdf)		
`CharacterPhysicsPath`	UI: browse for character .phys-file.		
`CharacterRigPath`	UI: browse for .rig files.		
`SkeletonPath`	UI: browse for .chr or .skel files.		
`JointName`	UI: list of character joints	`ICharacterInstance*`	
`AttachmentName`	UI: list of character attachments	`ICharacterInstance*`	
`SoundName`	UI: list of sounds		
`ParticleName`	UI: particle effect selection		
`Serialization/Decorators/Math.h`			
`RadiansAsDeg`	Edit or store radians as degrees	float, Vec3	
`Serialization/Decorators/Range.h`			
`Range`	Sets soft or hard limits for numeric values and provides a slider UI.	Numeric types	
`Serialization/Callback.h`			
`Callback`	Provides per-property callback function. See Adding callbacks to the PropertyTree (p. 247).	All types apart from compound ones (structs and containers)	

Decorator example

The following example uses the Range and CharacterPath decorators.

```
float scalar;
ar(Serialization::Range(scalar), 0.0f, 1.0f); // provides slider-UI
string filename;
ar(Serialization::CharacterPath(filename), "character", "Character"); // provides
 UI for file selection with character filter
```

Serialization context

The signature of the Serialize method is fixed. This can prevent the passing of additional arguments into nested Serialize methods. To resolve this issue, you can use a serialization context to pass a pointer of a specific type to nested Serialize calls, as in the following example.

```
void Scene::Serialize(Serialization::IArchive& ar)
{
  Serialization::SContext sceneContext(ar, this);
  ar(rootNode, "rootNode")
}

void Node::Serialize(Serialization::IArchive& ar)
{
  if (Scene* scene = ar.FindContext<Scene>())
  {
    // use scene
  }
}
```

Contexts are organized into linked lists. Nodes are stored on the stack within the SContext instance.

You can have multiple contexts. If you provide multiple instances of the same type, the innermost context will be retrieved.

You may also use contexts with the PropertyTree without modifying existing serialization code. The easiest way to do this is to use CContextList (QPropertyTree/ContextList.h), as in the following example.

```
// CContextList m_contextList;
tree = new QPropertyTree();
m_contextList.Update<Scene>(m_scenePointer);
tree->setArchiveContext(m_contextList.Tail());
tree->attach(Serialization::SStruct(node));
```

Serializing opaque data blocks

It is possible to treat a block of data in the archive in an opaque way. This capability enables the Editor to work with data formats it has no knowledge of.

These data blocks can be stored within Serialization::SBlackBox. SBlackBox can be serialized or deserialized as any other value. However, when you deserialize SBlackBox from a particular kind of archive, you must serialize by using a corresponding archive. For example, if you obtained your SBlackBox from JSONIArchive, you must save it by using JSONOArchive.

Adding callbacks to the PropertyTree

When you change a single property within the property tree, the whole attached object gets de-serialized. This means that all properties are updated even if only one was changed. This approach may seem wasteful, but has the following advantages:

- It removes the need to track the lifetime of nested properties, and the requirement that nested types be referenced from outside in safe manner.
- The content of the property tree is not static data, but rather the result of the function invocation. This allows the content to be completely dynamic. Because you do not have to track property lifetimes, you can serialize and de-serialize variables constructed on the stack.
- The removal of the tracking requirement results in a smaller amount of code.

Nevertheless, there are situations when it is desirable to know exactly which property changes. You can achieve this in two ways: 1) by using the `Serialize` method, or 2) by using `Serialization::Callback`.

1. Using the `Serialize` method, compare the new value with the previous value, as in the following example.

```
void Type::Serialize(IArchive& ar)
{
  float oldValue = value;
  ar(value, "value", "Value");
  if (ar.IsInput() && oldValue != value)
  {
 // handle change
  }
}
```

2. The second option is to use the `Serialization::Callback` decorator to add a callback function for one or more properties, as the following example illustrates.

```
#include <Serialization/Callback.h>
using Serialization::Callback;

ar(Callback(value,
            [](float newValue) { /* handle change */ }),
   "value", "Value");
```

Note
`Callback` works only with the PropertyTree, and should be used only in Editor code.

`Callback` can also be used together with other decorators, but in rather clumsy way, as the following example shows.

```
ar(Callback(value,
            [](float newValue) { /* handle change*/ },
            [](float& v) { return Range(v, 0.0f, 1.0f); }),
   "value", "Value");
```

Of the two approaches, the callback approach is more flexible, but it requires you to carefully track the lifetime of the objects that are used by the callback lambda or function.

PropertyTree in MFC window

If your code base still uses MFC, you can use the PropertyTree with it by using a wrapper that makes this possible, as the following example shows.

```
#include <IPropertyTree.h> // located in Editor/Include

int CMyWindow::OnCreate(LPCREATESTRUCT pCreateStruct)
{
  ...
  CRect clientRect;
  GetClientRect(clientRect);
  IPropertyTree* pPropertyTree = CreatePropertyTree(this, clientRect);
  ...
}
```

The `IPropertyTree` interface exposes the methods of `QPropertyTree` like `Attach`, `Detach` and `SetExpandLevels`.

Documentation and validation

`QPropertyTree` provides a way to add short documentation in the form of tool tips and basic validation.

The `Doc` method allows you to add tool tips to `QPropertyTree`, as in the following examples.

```
void IArchive::Doc(const char*)
```

```
void SProjectileParameter::Serialize(IArchive& ar)
{
  ar.Doc("Defines projectile physics.");

  ar(m_velocity, "velocity", "Velocity");
  ar.Doc("Defines initial velocity of the projectile.");
}
```

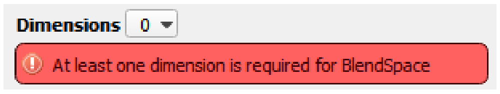
The `Doc` method adds a tool tip to last serialized element. When used at the beginning of the function, it adds the tool tip to the whole block.

The `Warning` and `Error` calls allow you to display warnings and error messages associated with specific property within the property tree, as in the following examples.

```
template<class T> void IArchive::Warning(T& instance, const char* format, ...)
template<class T> void IArchive::Error(T& instance, const char* format, ...)
```

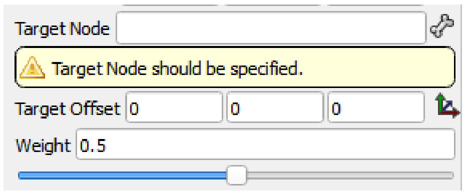
```
void BlendSpace::Serialize(IArchive& ar)
{
 ar(m_dimensions, "dimensions, "Dimensions");
 if (m_dimensions.empty())
  ar.Error(m_dimensions, "At least one dimension is required for BlendSpace");
}
```

The error message appears as follows.


Dimensions 0

At least one dimension is required for BlendSpace

Warning messages look like this:


Target Node

Target Node should be specified.

Target Offset 0 0 0

Weight 0.5

Drop-down menu with a dynamic list

If you want to specify an enumeration value, you can use the `enum` registration macro as described in the Defining data (p. 238) section.

There are two ways to define a drop-down menu: 1) transform your data into `Serialization::StringListValue`, or 2) implement a custom PropertyRow in the UI.

A short example of the first approach follows. The example uses a custom reference.

```
// a little decorator that would annotate string as a special reference
struct MyReference
{
 string& str;
 MyReference(string& str) : str(str) {}
};

inline bool Serialize(Serialization::IArchive& ar, MyReference& wrapper, const
 char* name, const char* label)
{
 if (ar.IsEdit())
 {
  Serialization::StringList items;
  items.push_back("");
  items.push_back("Item 1");
  items.push_back("Item 2");
  items.push_back("Item 3");
  Serialization::StringListValue dropDown(items, wrapper.str.c_str());
  if (!ar(dropDown, name, label))
   return false;
  if (ar.IsInput())
   wrapper.str = dropDown.c_str();
  return true;
 }
 else
 {
        // when loading from disk we are interested only in the string
  return ar(wrapper.str, name, label);
 }
}
```

Now you can construct `MyReference` on the stack within the `Serialize` method to serialize a string as a dropdown item, as in the following example.

```
struct SType
{
  string m_reference;
  void SType::Serialize(Serialization::IArchive& ar)
  {
 ar(MyReference(m_reference), "reference", "Reference");
  }
};
```

The second way to define a drop-down menu requires that you implement a custom PropertyRow in the UI. This takes more effort, but makes it possible to create the list of possible items entirely within editor code.

Demo and Video Capture

This section contains information on recording videos for benchmarking. Capturing audio and video is also discussed, using either the Perspective view of the Lumberyard Editor or in pure-game mode via the Launcher.

Topics

Capturing Video and Audio

This tutorial explains how to set up Lumberyard editor (or game) to capture video. Lumberyard outputs video as single frames. If required, it can also output stereo or 5.1 surround sound audio in `.wav` file format. You can edit the output with commonly available video editing software.

Preparation

Before you can start video and audio streams in preparation for capture, you must configure some settings that determine how the video will be captured. You configure these settings by using console commands. To save time, you can create configuration files that execute the necessary commands for you instead of typing the commands directly into the console. Example configuration files are presented later in this topic.

The next sections describe the settings and the console commands that configure them.

Video Settings

Frame Size and Resolution

The height and width of the captured frames in the editor is normally set to the exact view size of your rendered perspective window. To resize the view size, re-scale the perspective window, or right click in the top right of the perspective viewport where the frame size is displayed.

You can also capture higher than rendered images from Lumberyard Editor and Launcher.

The console variables that are now used in conjunction with Capture Frames are:

- `r_CustomResHeight=`*N* - Specifies the desired frame height in *N* pixels.
- `r_CustomResWidth=`*M* - Specifies the desired frame width in *M* pixels.

- `r_CustomResMaxSize=`*P* - Specifies the maximum resolution at which the engine will render the frames in *P* pixels.
- `r_CustomResPreview=`*R* - Specifies whether or how the preview is displayed in the viewport. Possible values for *R* are:

r_CustomResPreview	Preview status
0	No preview
1	Scaled to match the size of the viewport
2	Cropped to the size of the viewport

Frames Per Second

When deciding the number of frames per second to specify, keep in mind the following:

- NTSC standard video is approximately 30 frames per second, which is a good compromise between quality and file size.
- High quality video can have up to 60 frames per second, but the difference in quality of the increased number of frames is barely noticeable and can take up a lot of file space.
- Video at less than 24 FPS (a cinema standard) will not look smooth.

To specify a fixed frame rate, use the command:

```
t_fixedstep N
```

N specifies the time step. Time step is calculated by using the formula

```
step = 1 second/<number of frames>
```

A table of common time step values follows.

FPS	Time Step
25 (PAL)	0.04
30	0.033333333
60	0.0166666667

Video Capture File Format

You can capture pictures in several different file formats. A good choice for average quality is the `.jpeg` file format. The `.tga` or `.bmp` file formats are better for higher quality, and `.hdr` for pictures that use high-dynamic-range imaging.

To specify the capture file format, use the console command

```
capture_file_format N
```

N is `jpg`, `bmp`, `tga` or `hdr`.

Video Capture File Location

By default, recorded frames are stored in the folder *<root>*`\CaptureOutput`. To specify a custom folder, use the command:

```
capture_folder N
```

N is the name of the custom folder.

Caution
When you start a recording, the captured frames are placed in the currently specified folder and will overwrite existing files with the same name. To avoid losing work, create a folder for each recording, or move the existing files to another folder before you start.

Starting and Ending the Video Recording

After you have specified the values mentioned in the previous sections, you can start the recording by using the command:

```
capture_frames N
```

Setting *N* to **`1`** starts the recording, and setting *N* to **`0`** stops it.

Audio Settings

Before you begin, decide if you require audio in stereo or in 5.1 surround format, and then change your audio settings accordingly in the Windows control panel.

Deactivating the Sound System

After loading the level of your game that you want to capture, you must deactivate the sound system so that you can redirect the sound output to a file. To deactivate the sound system, use the command:

```
#Sound.DeactivateAudioDevice()
```

This redirects the sound output to a `.wav` file in the root folder of the game. The sound will not run in realtime, but be linked precisely to the time step that you set previously.

To write the sound capture, use the command:

```
s_OutputConfig N
```

Setting *N* to 3 activates the non-realtime writing of sound to the `.wav` file. Setting *N* to `0` specifies auto-detection (the default).

Reactivating the Sound System

To reset the sound system use the command:

```
#Sound.ActivateAudioDevice()
```

This creates a `.wav` file in the root folder of the game. The file will continue to be written to until you run the following combination of commands to deactivate the audio device:

```
#Sound.DeactivateAudioDevice()
```

```
s_OutputConfig 0
```

```
#Sound.ActivateAudioDevice()
```

Tip
Although these commands reset the sound system, some sounds won't start until they are correctly triggered again. This applies particularly to looped sounds. To get looped sounds to play, start the recording of video and sound first, and then enter any area that triggers the looped sounds that you want to record.

Configuration Files

Creating Configuration Files

- To ensure that multiple recordings use exactly the same settings, create a configuration file that you can use for each of them. This will ensure that all of your captured files have the same format.

  An example configuration file:

  ```
  sys_spec = 4
  Fixed_time_step 0.0333333333
  Capture_file_format jpg
  Capture_folder myrecording
  r_width 1280
  r_height 800
  ```

  The command `sys_spec = 4` sets the game graphic settings to "very high" to generate the best appearance.
- To speed up the process of starting and stopping the recording, you can create two configuration files: one to start the video, and one to stop it.
  - To start recording, use a config file like the following:

    ```
    #Sound.DeactivateAudioDevice()
    s_OutputConfig 3
    #Sound.ActivateAudioDevice()
    Capture_frames 1
    ```

  - To stop recording, use a config file like the following:

```
Capture_frames 0
#Sound.DeactivateAudioDevice()
s_OutputConfig 0
#Sound.ActivateAudioDevice()
```

Executing the Config Files

To run the config file, open the console and type the following command:

```
Exec N
```

N is the name of the config file.

Recording Time Demos

Overview

Lumberyard Editor can record and play back player input and camera movement.

Note
Recording of some player actions such as vehicle movement are not supported.

To use the feature, you must start game mode in Lumberyard Editor and then record in it. To start game mode, press **`Ctrl + G`** after a level has been fully loaded, or load the level in pure-game mode.

Output like the following appears both in the console and in the `timedemo.log` file in the directory corresponding to the level used:

```
TimeDemo Run 131 Finished.
Play Time: 3.96s, Average FPS: 50.48
Min FPS: 0.63 at frame 117, Max FPS: 69.84 at frame 189
Average Tri/Sec: 14037316, Tri/Frame: 278071
Recorded/Played Tris ratio: 0.99
```

Recording Controls

Command	Keystroke	Console Commands
Start Recording	**`Ctrl + PrintScreen`**	`record`
End Recording	**`Ctrl + Break`**	`stoprecording`
Start Playback	**`Shift + PrintScreen`**	`demo`
Stop Playback	**`Ctrl + Break`**	`stopdemo`

Related Console Variables

- `stopdemo` - Stop playing a time demo.

- `demo` *`demoname`* - Play the time demo from the file specified
- `demo_fixed_timestep` - Specify the number of updates per second.
- `demo_panoramic` - Use a anoramic view when playing back the demo.
- `demo_restart_level` *`N`* - Restart the level after each loop. Possible values for *`N`*: `0` = Off; `1` = use quicksave on first playback; 2 = load level start.
- `demo_ai`: Enable or disable AI during the demo.
- `demo_savestats` - Save level stats at the end of the loop.
- `demo_max_frames` - Specify the maximum number of frames to save.
- `demo_screenshot_frame` *`N`* - Make a screenshot of the specified frame during demo playback. If a negative value for *`N`* is supplied, takes a screenshot every *`N`* frame.
- `demo_quit`: Quit the game after the demo run finishes.
- `demo_noinfo` - Disable the information display during demo playback.
- `demo_scroll_pause` - Enable the use of the **`ScrollLock`** key to pause demo play and record.
- `demo_num_runs` - Specify the number of times to loop the time demo.
- `demo_profile`: Enable demo profiling.
- `demo_file` - Specify the time demo filename.

Entity System

This section covers topics related to the Entity system. Entities are objects, placed inside a level, that players can interact with.

This section includes the following topics:

Entity Property Prefixes

The Lumberyard Editor supports typed properties where the type is derived from special prefixes in the property name. For a complete list of supported prefixes, refer to the `s_paramTypes` array, defined in `Objects/EntityScript.cpp`. This array maps prefixes to variable types.

The following prefixes are supported by Lumberyard:

```
 { "n",                  IVariable::INT, IVariable::DT_SIMPLE, SCRIPTPARAM_POS
ITIVE },
 { "i",                   IVariable::INT, IVariable::DT_SIMPLE,0 },
 { "b",                   IVariable::BOOL, IVariable::DT_SIMPLE,0 },
 { "f",                   IVariable::FLOAT, IVariable::DT_SIMPLE,0 },
 { "s",                   IVariable::STRING, IVariable::DT_SIMPLE,0 },

 { "ei",                  IVariable::INT, IVariable::DT_UIENUM,0 },
 { "es",                  IVariable::STRING, IVariable::DT_UIENUM,0 },

 { "shader",              IVariable::STRING, IVariable::DT_SHADER,0 },
 { "clr",                 IVariable::VECTOR, IVariable::DT_COLOR,0 },
 { "color",               IVariable::VECTOR, IVariable::DT_COLOR,0 },

 { "vector",              IVariable::VECTOR, IVariable::DT_SIMPLE,0 },
 { "snd",                  IVariable::STRING, IVariable::DT_SOUND,0 },
 { "sound",                IVariable::STRING, IVariable::DT_SOUND,0 },
 { "dialog",               IVariable::STRING, IVariable::DT_DIALOG,0 },

 { "tex",                  IVariable::STRING, IVariable::DT_TEXTURE,0 },
 { "texture",              IVariable::STRING, IVariable::DT_TEXTURE,0 },

 { "obj",                  IVariable::STRING, IVariable::DT_OBJECT,0 },
 { "object",               IVariable::STRING, IVariable::DT_OBJECT,0 },

 { "file",                 IVariable::STRING, IVariable::DT_FILE,0 },
 { "aibehavior",           IVariable::STRING, IVariable::DT_AI_BEHAVIOR,0 },
 { "aicharacter",          IVariable::STRING, IVariable::DT_AI_CHARACTER,0 },
 { "aipfpropertieslist", IVariable::STRING, IVariable::DT_AI_PFPROPERTIESLIST,0
 },
 { "aiterritory",          IVariable::STRING, IVariable::DT_AITERRITORY,0 },
 { "aiwave",               IVariable::STRING, IVariable::DT_AIWAVE,0 },

 { "text",                 IVariable::STRING, IVariable::DT_LOCAL_STRING,0 },
 { "equip",                IVariable::STRING, IVariable::DT_EQUIP,0 },
 { "reverbpreset",         IVariable::STRING, IVariable::DT_REVERBPRESET,0 },
 { "eaxpreset",            IVariable::STRING, IVariable::DT_REVERBPRESET,0 },

 { "aianchor",             IVariable::STRING, IVariable::DT_AI_ANCHOR,0 },

 { "soclass",              IVariable::STRING, IVariable::DT_SOCLASS,0 },
 { "soclasses",            IVariable::STRING, IVariable::DT_SOCLASSES,0 },
 { "sostate",              IVariable::STRING, IVariable::DT_SOSTATE,0 },
 { "sostates",             IVariable::STRING, IVariable::DT_SOSTATES,0 },
 { "sopattern",            IVariable::STRING, IVariable::DT_SOSTATEPATTERN,0 },
 { "soaction",             IVariable::STRING, IVariable::DT_SOACTION,0 },
 { "sohelper",             IVariable::STRING, IVariable::DT_SOHELPER,0 },
 { "sonavhelper",          IVariable::STRING, IVariable::DT_SONAVHELPER,0 },
 { "soanimhelper",         IVariable::STRING, IVariable::DT_SOANIMHELPER,0 },
 { "soevent",              IVariable::STRING, IVariable::DT_SOEVENT,0 },
 { "sotemplate",           IVariable::STRING, IVariable::DT_SOTEMPLATE,0 },
 { "gametoken",            IVariable::STRING, IVariable::DT_GAMETOKEN, 0 },
 { "seq_",                 IVariable::STRING, IVariable::DT_SEQUENCE, 0 },
 { "mission_",             IVariable::STRING, IVariable::DT_MISSIONOBJ, 0 },
```

Creating a New Entity Class

The following example creates an entity class called **Fan**.

- Create a new entity definition file with the extension ".ent", for example "`GameSDK\Entities\Fan.ent`". This file will expose the entity to the engine.

```
<Entity
  Name="Fan"
  Script="Scripts/Entities/Fan.lua"
/>
```

- Create a new Lua script file, for example `GameSDK\Entities\Scripts\Fan.lua`. The Lua file will define the entity logic.

```
Fan = {
  type = "Fan",                                   -- can be useful for
scripting

  -- instance member variables
  minrotspeed = 0,
  maxrotspeed = 1300,
  acceleration = 300,
  currrotspeed = 0,
  changespeed = 0,
  currangle = 0,

  -- following entries become automatically exposed to the editor and serial
ized (load/save)
  -- type is defined by the prefix (for more prefix types, search for s_para
mTypes in /Editor/Objects/EntityScript.cpp)
  Properties = {
  bName = 0,                              -- boolean example, 0/1
  fName = 1.2,                            -- float example
  soundName = "",                         -- sound example
  fileModelName = "Objects/box.cgf",      -- file model
  },

  -- optional editor information
  Editor = {
  Model = "Editor/Objects/Particles.cgf",   -- optional 3d object that repres
ents this object in editor
  Icon = "Clouds.bmp",                    -- optional 2d icon that represents
 this object in editor
  },
}

-- optional.  Called only once on loading a level.
-- Consider calling self:OnReset(not System.IsEditor()); here
function Fan:OnInit()
  self:SetName( "Fan" );
  self:LoadObject( "Objects/Indoor/Fan.cgf", 0, 0 );
  self:DrawObject( 0, 1 );
end

-- OnReset() is usually called only from the Editor, so we also need OnInit()
-- Note the parameter
function Fan:OnReset(bGameStarts)
end

-- optional.  To start having this callback called, activate the entity:
-- self:Activate(1); -- Turn on OnUpdate() callback
function Fan:OnUpdate(dt)
  if ( self.changespeed == 0 ) then
  self.currrotspeed = self.currrotspeed - System.GetFrameTime() * self.accel
eration;
  if ( self.currrotspeed < self.minrotspeed ) then
  self.currrotspeed = self.minrotspeed;
  end
  else
  self.currrotspeed = self.currrotspeed + System.GetFrameTime() * self.accel
eration;
  if ( self.currrotspeed > self.maxrotspeed ) then
  self.currrotspeed = self.maxrotspeed;
  end
  end
  self.currangle = self.currangle + System.GetFrameTime() * self.currrotspeed;

  local a = { x=0, y=0, z=-self.currangle };
  self:SetAngles( a );
end

-- optional serialization
function Fan:OnSave(tbl)
 tbl.currangle = self.currangle;
end

-- optional serialization
function Fan:OnLoad(tbl)
 self.currangle = tbl.currangle;
end

-- optional
function Fan:OnSpawn()
end

-- optional
function Fan:OnDestroy()
end

-- optional
function Fan:OnShutDown()
end

-- optional
function Fan:OnActivate()
  self.changespeed = 1 - self.changespeed;
end
```

Entity Pool System

The topics in this section describe the entity pool system, including how it is implemented, how to register a new entity class to be pooled, and how to debug it. For more information on using entity pools in the Lumberyard Editor, see the Lumberyard User Guide.

This section includes the following topics:

- Debugging Utilities (p. 269)

The following processes must take place when creating an entity pool and preparing it for use. Each of these processes is described in more detail.

1. An entity pool is created by using the information in an entity pool definition.
2. An entity pool is populated with entity containers.
3. An entity pool is validated by testing the entity pool signature of one of the entity containers against the entity pool signature of each `Entity` class mapped to the pool.
4. All entities marked to be created through the pool have an entity pool bookmark created for them.
5. An entity pool bookmark is prepared from or returned to the entity pool, which is mapped to its `Entity` class on demand.

Editor Usage

When running in the Lumberyard Editor, the entity pool system is not fully enabled. All entities are created outside the pools when playing in-game in the Editor. However, all flow node actions with entity pools will still work in the Lumberyard Editor, mimicking the final results that you will see in-game.

Note
The entity pool listeners `OnEntityPreparedFromPool` and `OnEntityReturnedToPool` are still called in the Editor, even though the entity itself is not removed/reused.

Static versus Dynamic Entities

Entities can be either static or dynamic. A static entity is placed in the Editor and exported with the level. This entity always exists. A property associated with the exported information determines whether it should be pooled (and not created during level load) or instead have an entity pool bookmark made for it. A dynamic entity is created at run-time, usually from game code. The information is constructed at run-time, usually just before it is created, and passed on to the Entity system for handling. This information also indicates whether or not it should go through an entity pool.

Entity Pool Definitions

Entity pools must be defined in the file `\Game\Scripts\Entities\EntityPoolDefinitons.xml`. An entity pool definition is responsible for defining the following:

- the empty class that will be used by entity containers when they're not in use
- the entity classes mapped to the pool
- other properties that describe the pool and how it is used.

In general, a pool is initially filled with a defined number of entity containers; that is, empty `CEntity` classes with all the required entity proxies and game object extensions that are normally created when an entity belonging to an entity class mapped to the definition is fully instantiated. For example, a normal AI entity will have the following entity proxies: sound extension, script extension, render extension, and the game object as its user extension; as its game object extension, it will have the `CPlayer` class. All of these classes are instantiated for each empty `CEntity` instance, and is reused by the entities as they are created from the pool.

The following illustrates an entity pool definition:

```
EntityPoolDefinitions.xml

<Definition name="AI" emptyClass="NullAI" maxSize="16" hasAI="1" defaultBook
marked="0" forcedBookmarked="0">
    <Contains>
        <Class>Grunt</Class>
        <Class>Flyer</Class>
    </Contains>
</Definition>
```

Empty Class

The empty class is defined using the `emptyClass` attribute, which takes the name of a valid entity class. The purpose of the empty class is to:

- satisfy the engine's requirement to have an entity class associated with an entity at all times; an empty container is initialized/reused to this entity class
- prepare all needed entity proxies and game object extensions needed for the entities

For example, building on the definition shown in the previous section, you would create an empty class called "NullAI" and register it the same way as the other AI classes above. Then:

1. Declare the entity class and map it to its Lua script via the game factory.

   ```
   GameFactory.cpp

   REGISTER_FACTORY(pFramework, "NullAI", CPlayer, true);
   ```

2. Create the Lua script for it. View sample code at `\Game\Scripts\Entities\AI\NullAI.lua`.

These steps will allow Lumberyard to see "NullAI" as a valid entity class. In addition, by mapping `CPlayer` to it, you ensure that the correct game object extension is instantiated for the entity containers. The Lua script needs to create all the entity proxies for the entity containers. In the sample code, a render proxy is created, even though we aren't loading an asset model for this entity. For more details, see the discussion of entity pool signatures in Entity Pool Creation (p. 263).

Entity Class Mapping

In an entity pool definition file, the `<Contains>` section should include maps to all the entity classes that an entity must belong to when it is created through this pool. You can map as many as you want by adding a new `<Class>` node within this section. It is important that each entity have the same dynamic class hierarchy as the empty class when fully instantiated. See Debugging Utilities (p. 269)for useful debugging tools to verify that this is the case.

Other Properties

An entity pool definition can define the following additional properties.

name
: Unique identity given to an entity pool, useful for debugging purposes. The name should be unique across all definitions.

maxSize

Largest pool size this pool can reach. By default, this is also the number of entity containers created to fill the pool when loading a level. This value can be overwritten for a level by including an `EntityPools.xml` file inside the level's root folder. This file can only be used to decrease the number of entity containers created per pool; it cannot exceed the `maxSize` value defined here. This is useful when you need to reduce the memory footprint of the entity pools per level. The following example file adjusts the size of an AI entity pool to "2".

```
LevelEntityPools.xml

<EntityPools>
    <AI count="2" />
</EntityPools>
```

hasAI

Boolean value that indicates whether or not the entity pool will contain entities that have AI associated with them. It is important to set this property to TRUE if you are pooling entities with AI.

defaultBookmarked

Boolean value that indicates whether or not an entity belonging to one of the entity classes mapped to this pool is flagged as "created through pool" (see Creating and Destroying Static Entities with Pools (p. 264)). This flag determines whether or not, during a level load, an entity pool bookmark is created for the entity instead of being instantiated.

forcedBookmarked

Boolean value that indicates whether or not an entity belonging to one of the entity classes mapped to this pool must be created through the pool. This property overrides an entity's "created through pool" flag (see Creating and Destroying Static Entities with Pools (p. 264)).

Entity Pool Creation

When loading a level, an entity pool is created for each entity pool definition. On creation, the pool is filled with empty containers (instances of `CEntity` using the `emptyClass` attribute value as the entity class. These empty containers come with some expectations that must be satisfied:

- Containers should be minimal in size. This means you should not load any assets or large amounts of data into them. For example, in the sample Lua script (`\Game\Scripts\Entities\AI\NullAI.lua`), the NullAI entity does not define a character model, animation graph, body damage definition, etc.
- Containers should have the same entity proxies and game object extensions created for them as compared to a `CEntity` fully instantiated using each of the mapped entity classes.

Once the pool is created, an entity pool signature is generated using one of the empty containers. An entity pool's signature is a simple container that maps the dynamic class hierarchy of an entity.

One of the functions of the entity pool system is to avoid as much as possible dynamic allocation for delegate classes used by entities. Key examples of these are the entity proxies and game object extensions used by entities. When an entity pool's empty containers are first created, the delegate classes that will be used by the real entities contained in them are also supposed to be created. To ensure that this is the case, the entity pool signature is used. It works as follows:

1. A `TSerialize` writer is created. It is passed to each entity proxy and game object extension that exists in the entity.
2. Each proxy and extension is expected to write some info to the `TSerialize` writer. This information should be unique.
3. Two signatures can then be compared to see if they contain the same written information, verifying they contain the same dynamic class hierarchy.

All of the entity proxies have already been set up to write their information to the `TSerialize` writer. However, if you create a new game object extension (or a new entity proxy), then you will need to set the class up to respond to the Signature helper when needed. To do this, implement the virtual method (Entity Proxy: `GetSignature`; Game Object Extension: `GetEntityPoolSignature`) and write information about the class to the `TSerialize` writer. Generally, all that is needed is to just begin/end a group with the class name. The function should then return TRUE to mark that the signature is valid thus far.

`CActor::GetEntityPoolSignature Example`

```
bool CActor::GetEntityPoolSignature( TSerialize signature )
{
 signature.BeginGroup("Actor");
 signature.EndGroup();
 return true;
}
```

The section Debugging Utilities (p. 269) discusses how to view the results of entity pool signature tests in order to verify that everything is working as expected.

Creating and Destroying Static Entities with Pools

This topic covers issues related to handling static entities.

Entity Pool Bookmarks

When an entity is marked to be created through the pool, it is not instantiated during the level load process. Instead, an entity pool bookmark is generated for it. The bookmark contains several items:

- Entity ID reserved for the entity, assigned when the level was exported. You will use this entity ID later to tell the system to create the entity.
- Static instanced data that makes the entity unique. This includes the `<EntityInfo>` section from the `mission.xml` file, which contains area information, flow graph information, child/parent links, PropertiesInstance table, etc.
- Serialized state of the entity if it has been returned to the pool in the past. See more details in Serialization (p. 267).

In each entity's `<EntityInfo>` section in the `mission.xml` file (generated when the level is exported from the Editor), there's a `CreatedThroughPool` property. This property can be referenced from the `SEntitySpawnParams` struct. If set to TRUE, the `EntityLoadManager` module will **not** create a `CEntity` instance for the entity. Instead, it will delegate the static instanced data and reserved entity ID to the `EntityPoolManager` to create a bookmark.

`CEntityLoadManager::ParseEntities`

```
SEntityLoadParams loadParams;
if (ExtractEntityLoadParams(entityNode, loadParams))
{
    if (bEnablePoolUse && loadParams.spawnParams.bCreatedThroughPool)
    {
        CEntityPoolManager *pPoolManager = m_pEntitySystem->GetEntityPoolMan
ager();
       bSuccess = (pPoolManager && pPoolManager->AddPoolBookmark(loadParams));

    }
    // Default to creating the entity
    if (!bSuccess)
    {
        EntityId usingId = 0;
        bSuccess = CreateEntity(loadParams, usingId);
    }
}
```

Preparing a Static Entity

To prepare a static entity, call `IEntityPoolManager::PrepareFromPool`, passing in the entity ID associated with the static entity you want to create. In response, the following execution flow takes place:

1. System determines if the request can be processed in this frame. It will attempt to queue up multiple requests per frame and spread them out. If the parameter `bPrepareNow` is set to TRUE or if no prepare requests have been handled this frame, the request will be handled immediately. Otherwise, it will be added to the queue. Inside `CEntityPoolManager::LoadBookmarkedFromPool`, the EntityLoadManager is requested to create the entity.

   Note
   Note: If this activity is happening in the Editor, the entity will simply have its Enable event called. This will mimic enabling the entity via Flow Graph (unhide it). In this situation, the execution flow skips to the final step.

2. System searches for an entity container (either empty, or still in use) to hold the requested entity. The function `CEntityPoolManager::GetPoolEntity` looks through the active entity pools to find one that contains the entity class of the given static entity. Once the correct pool is found, the container is retrieved from it. The actual order is as follows:
   a. If a `forcedPoolId` (entity ID of one of the empty containers created to populate the pool) is requested, find that entity container and return it.
   b. If no `forcedPoolId` is requested, get an entity container from the inactive set (entity containers not currently in use).
   c. If no inactive containers are available, get one from the active set (entity containers currently in use). This action uses a "weight" value to determine which container to return. A special Lua function in the script is used to request weights for each empty container (`CEntityPoolManager::GetReturnToPoolWeight`). A negative weight means it should not be used at all if possible. The system might pass in an urgent flag, which means the pool is at its maximum size.
   d. If an empty container can still not be found, an urgent flag will be ignored and the system will try to grow the pool. This is only possible if the pool was not created at its maximum size (this happens when the maximum pool size is overridden for a level with a smaller maximum size). In this case, a new entity container is generated, added to the pool, and immediately used.
3. The retrieved entity container, along with the static instanced data and reserved entity ID gathered from its bookmark, is passed on through the function`CEntityLoadManager::CreateEntity`, which begins the Reload process. `CreateEntity` uses the provided entity container instead of creating a new `CEntity` instance. It will handle calling the Reload pipeline on the entity container, and then install all the static instanced data for the prepared static entity. The Reload pipeline is as follows:
   a. The function `CEntity::ReloadEntity` is called on the entity container. The `CEntity` instance will clean itself up internally and begin using the static instanced data of the entity being prepared. The Lua script also performs cleanup using the function `OnBeingReused`.
   b. The Entity system's salt buffer and other internal containers are updated to reflect that this entity container now holds the reserved entity ID and can be retrieved using it.
   c. Entity proxies are prompted to reload using the static instanced data provided. This is done by calling `IEntityProxy::Reload`; each proxy is expected to correctly reset itself with the new data provided.

The Script proxy is always the first to be reloaded so that the Lua script can be correctly associated before the other proxies attempt to use it.

If the game object is being used as the User proxy, all the game object extensions for the container are also prompted to reload. This is done by calling `IGameObjectExtension::ReloadExtension` on all extensions. If this function returns FALSE, the extension will be deleted. Once this is done, `IGameObjectExtension::PostReloadExtension` is called on all extensions. This behavior mimics the Init and PostInit logic. Each extension is expected to correctly reset itself with the new data provided.

4. If any serialized data exists within the bookmark, the entity container is loaded with that data. This ensures that the static entity resumes the state it was in last time it was returned to the pool. This process is skipped if this is the first time the static entity is being prepared.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` will return the entity container that is now housing the static entity and its information.

Returning a Static Entity to the Pool

To return a static entity, call the function `IEntityPoolManager::ReturnToPool`. You must pass in the entity ID associated with the static entity. In response, the following execution flow takes place:

1. The function `CEntityPoolManager::ReturnToPool` finds the bookmark and the entity pool containing the current entity container housing the static entity.
2. Depending on the `bSaveState` argument, the `CEntity` instance is (saved) and its serialized information is added to the bookmark. This ensures that if the static entity is prepared again later, it will resume its current state.
3. The entity container goes through the Reload process again. This time, however, the entity container is reloaded using its empty class, effectively removing all references to loaded assets/content and put it back into a minimal state.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` to find the static entity will return NULL.

Creating and Destroying Dynamic Entities with Pools

The processes for creating and destroying dynamic entities are similar to those for static entities, which one key exception: dynamic entities have no entity pool bookmarks (at least initially). Because they are not exported in the level, they have no static instanced data associated with them and so no bookmark is made for them.

Creating a Dynamic Entity

As with static entities, creating a dynamic entity with the pool starts with calling `IEntitySystem::SpawnEntity`. Construct an `SEntitySpawnParams` instance to describe its static instanced data. When filling in this struct, set the `bCreatedThroughPool` property to TRUE if you wish to have the entity be created through the pool. In the following example, a vehicle part from the Vehicle system is being spawned through the pool:

```
SEntitySpawnParams spawnParams;
spawnParams.sName = pPartName
spawnParams.pClass = gEnv->pEntitySystem->GetClassRegistry()->Find
Class("VehiclePartDetached");
```

```
spawnParams.nFlags = ENTITY_FLAG_CLIENT_ONLY;

spawnParams.bCreatedThroughPool = true;

IEntity* pSpawnedDebris = gEnv->pEntitySystem->SpawnEntity(spawnParams);
```

Once `SpawnEntity`, the following execution flow takes place:

1. `CEntitySystem::SpawnEntity` will check for an entity pool associated with the provided entity class. If so, it will delegate the workload to the entity pool manager.
2. From within `CEntityPoolManager::PrepareDynamicFromPool`, an entity pool bookmark is created for the new entity. This is done primarily for serialization purposes.
3. The execution flow follows the same sequence as preparing a static entity (see Creating and Destroying Static Entities with Pools (p. 264).
4. If the process is successful, the entity container now housing the information is returned. Otherwise, `SpawnEntity` creates a new `CEntity` instance to satisfy the request.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` will return the entity container now housing the dynamic entity and its information.

Destroying a Dynamic Entity with the Pool

As with static entities, use `IEntitySystem::RemoveEntity` or any other method that can destroy an entity. The entity pool manager will return the entity container to the pool, freeing it for use elsewhere and removing the dynamic entity in the process. The resulting execution flow differ from destroying static entities in two ways:

- Dynamic entities are not serialized when they are returned.
- The entity pool bookmark associated with the dynamic entity is removed. It is no longer needed.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` will return NULL.

Serialization

All entities created or prepared through the entity pool system are serialized by the system for game save/load. For this reason, do not serialize those entities marked as coming from the pool (`IEntity:IsFromPool`) in your normal serialization. This is handled in Lumberyard's default implementation for saving and loading the game state.

The entity pool system is serialized from the Entity system's implementation of the `Serialize` function.

Saving Entity Pools

The following process occurs when the game state is being saved:

1. All active entity containers in all entity pools are updated. This results in `CEntityPoolManager::UpdatePoolBookmark` being called for each active entity container. As long as the entity does not have the ENTITY_FLAG_NO_SAVE flag set on it, the bookmark is serialized as follows:
   a. Serialize Helper writes to the bookmark's `pLastState` (an `ISerializedObject`), which contains the serialized state of the entity.

   b. The callback `CEntityPoolManager::OnBookmarkEntitySerialize` runs through the serialization process on the entity. This ensures that the general information, properties and all entity proxies are serialized using their overloaded `Serialize()` implementation.
   c. Any listeners subscribed to the `OnBookmarkEntitySerialize` callback are able to write data into the bookmark at this time. This is used to also bookmark AI objects along with the entity.
2. All entity pool bookmarks are saved, including the static entity and dynamic entity usage counts.
3. If any prepare requests are currently queued, the prepare request queue is saved.

Loading Entity Pools

The following process occurs when the game state is being loaded:

1. The saved entity pool bookmarks are read in. If the bookmark is marked as containing a dynamic entity, it is read to ensure it exists. Each bookmark's `pLastState` is read in and updated.
2. If the entity pool bookmark contains an entity that was active at the time the game was saved, the entity is created/prepared from the pool once more.
   a. While the entity is being created/prepared, it will load its internal state using the `pLastState` at its final step, because the object contains information at this point.
   b. This will also call the `OnBookmarkEntitySerialize` listener callback, allowing other systems to read data from the bookmark.

Listener/Event Registration

There are several listener and various event callbacks dealing with entity pool usage. These callbacks are important for sub-systems that rely on entity registration. They can notify you when an entity has been prepared or returned to the pool so that you can register and unregister it with your subsystems as needed.

IEntityPoolListener

This listener can be subscribed to via `IEntityPoolManager::AddListener`. It contains the following callbacks:

OnPoolBookmarkCreated
  Called when an entity pool bookmark has been created. The reserved entity ID for the pooled entity is passed in, along with the static instanced data belonging to it.

OnEntityPreparedFromPool
  Called when an entity (static or dynamic) has been prepared from the pool. You are given both the entity ID and the entity container that is now housing the entity. This is called at the end of the prepare entity process.

OnEntityReturnedToPool
  Called when an entity (static or dynamic) has been returned to the pool. You are given both the entity ID and the entity container that is currently housing the entity. This is called at the start of the return entity process.

OnPoolDefinitionsLoaded
  Called at initialization, with information allowing listeners to set up their own resources for working with the pool. Currently passes the total number of pooled entities that have AI attached.

OnBookmarkEntitySerialize
  Called during reads and writes from entity bookmarks, allowing listeners to store additional data in the bookmark.

IEntitySystemSink

This listener has a special callback, OnReused, that notifies you when an entity has been reloaded. This is the process an entity container goes through when a static entity is being prepared into it, or a dynamic entity is being created inside it. You are given the entity container that houses the entity as well as the static instanced data belonging to it.

Debugging Utilities

There are several debugging utilities you can use to manage the entity pools and see how they are being used during gameplay.

Debugging Entity Pool Bookmarks

To see the status of all entity pool bookmarks that currently exist during the game, use the following console command.

```
es_dump_bookmarks [filterName] [dumpToDisk]
```

This command causes text to be written to the console and game log file for every bookmark requested.

Arguments

filterName
(Optional) Allows you to filter your request to get bookmarks only for entities whose names contain the specified value as a substring. To display all bookmarks, set this argument to "all" or leave it empty.

dumpToDisk
(Optional) Allows you to output to disk all static instanced data associated with the displayed bookmarks. If supplied and its a non-zero numerical file, data will be stored at `\User\Bookmarks\LevelName\EntityName.xml`.

Data displayed

The following information is displayed for each bookmark:

- Name of the bookmarked entity.
- Layer the bookmarked entity belongs to.
- Entity class name the bookmarked entity uses.
- Reserved entity ID associated with the bookmarked entity.
- If the bookmarked entity has the No Save Entity Flag associated with it.
- If the bookmarked entity is static or dynamic.
- If the bookmarked entity contains any serialized data (and the memory footprint of the information if available).
- If the bookmarked entity contains any static instanced data (and the memory footprint of the information if available).

Entity ID Explained

When referring to a dynamic C++ object, pointers and reference counting can be used, but a better method is to use a weak reference that allows you to remove an object and have all references become invalid.

This option limits the need to iterate over all objects to invalidate objects being removed, which results in performance costs.

With each reference, Lumberyard stores a number called the "salt" (also called a "magic number"). This number, together with the **index**, gives the object a unique entity ID over the game lifetime. Whenever an object is destroyed and the index is reused, the salt is increased and all references with the same index become invalid. To get an entity position/pointer, the entity manager needs to resolve the entity ID; as the salt is different, the method fails.

The class `CSaltBufferArray` handles adding and removing objects and does the required adjustments to the salt. The object array is kept compact for more cache-friendly memory access. Storing `EntityId` references to disc is possible and used for saved games and by the Editor game export. However, when loading a saved game of a level that has been patched and now has more entities, this can result in a severe conflict. To solve this problem, dynamic entities are created starting with a high index counting down, while static entities are created starting with a low index counting up.

Entity IDs have the following limitations:

- A 16-bit index allows up to approximately 65,000 living objects. This should be enough for any non-massive multiplayer game. In a massive multiplayer game, the method described here should not be used by the server. However, it can be used between specific clients and the server.
- A 16-bit salt value allows a game to reuse an index up to approximately 65,000 times. If that happens, the index can no longer be used. This should be enough for any non-massive multiplayer game, when used with some care—don't create and destroy objects (such as bullets) too rapidly. A massive multiplayer game, or any game that supports multi-day game sessions, can run into this limit.

Adding Usable Support on an Entity

Overview

Players may be able to interact with an entity using a key press ('F' by default). Entities that can be interacted with will be enabled with a special on-screen icon inside the game to inform the player that interaction is possible.

To use this feature, you need to create a script that implements two functions: `IsUsable()` and `OnUsed()`.

Preparing the Script

The script should look like this:

```
MakeUsable(NewEntity)

function NewEntity:IsUsable(userId)
  -- code implementation
  return index;
end

function NewEntity:OnUsed(userId, index)
  -- code implementation
end
```

Implementing IsUsable

The `IsUsable()` function is called when a player is aiming the cross-hairs towards the entity. The function will determine if the entity can be interacted with by the player doing the aiming. The function only accepts a single parameter: the player's entity ID.

If the player cannot interact with the entity, the function should return `0`. This value causes the UI to not render the "USE" icon over the entity.

If the player can interact with the entity, the function should return a positive value. This value will be stored and later used when calling the `OnUsed()` function.

Implementing OnUsed

The `OnUsed()` function is called when a player presses interacts with the entity (such as by pressing the Use key when the USE icon is active. This function accepts two parameters: (1) the player's entity ID, and (2) the value returned by `IsUsable()`.

Entity Scripting

This section contains topics on using Lua scripting to work with the Entity system.

This section includes the following topics:

Structure of a Script Entity

To implement a new entity using Lua, two files need to be created and stored in the game folder:

- The Ent file tells the Entity system the location of the Lua script file.
- The Lua script file implements the desired properties and functions.

With the SDK, both the `.ent` and `.lua` files are stored inside the `<Game_Folder>\Scripts.pak` file.

Ent File

The Ent files are all stored inside the `<Game_Folder>\Entities` directory and need to have the `.ent` file extension. The content is XML as follows:

```
<Entity
 Name="LivingEntity"
 Script="Scripts/Entities/Physics/LivingEntity.lua"
/>
```

Entity properties set in the Ent file include:

Name
Name of the entity class.

Script
Path to the Lua script that implements the entity class.

Invisible
Flag indicating whether or not the entity class is visible in Lumberyard Editor.

Lua Script

The Lua script, in addition to implementing the entity class, provides a set of information used by Lumberyard Editor when working with entities on a level. The property values set inside the Lua script are default values assigned to new entity instances. Editor variables specify how entities are drawn in Lumberyard Editor.

The following code excerpt is from the sample project files in your Lumberyard directory (`.../dev/cache/samplesproject/pc/samplesproject/scripts/entities/physics/livingentity.lua`).

```
LivingEntity = {
 Properties = {
  soclasses_SmartObjectClass = "",
  bMissionCritical = 0,
  bCanTriggerAreas = 1,
  object_Model = "objects/default/primitive_capsule.cgf",
  bExcludeCover=0,

  Physics = {
   bPhysicalize = 1, -- True if object should be physicalized at all.
   bPushableByPlayers = 1,
  },

  MultiplayerOptions = {
   bNetworked  = 0,
  },
 },
 ...
 Editor={
  Icon = "physicsobject.bmp",
  IconOnTop=1,
 },
}
```

This information is followed by functions that implement the entity class.

Properties

Entity properties are placed inside the entity class. These properties are assigned to all new instances of the entity class created, visible and editable in Lumberyard Editor as the instance's **Entity Properties** table. The property values set for individual entity instances placed on a level are saved in the level file. When a property of an entity instance is changed in Lumberyard Editor, the `OnPropertyChange()` function called (if it is has been implemented for the script entity).

Lumberyard Editor provides the Archetype tool for assigning a common set of properties reused for multiple instance (even across multiple levels). More information on Archetypes can be found in the Lumberyard User Guide.

When specifying entity class property names, use the following prefixes to signal the data type expected for a property value. This enables Lumberyard Editor to validate a property value when set.

Entity class property prefixes

Prefix	Data Type
b	boolean
f	float
i	integer
s	string
clr	color
object_	object compatible with Lumberyard (CFG, CGA, CHR or CDF file)

You can add special comments to property values that can be utilized by the engine. For example:

```
--[25,100,0.1,"Damage threshold"]
```

This comment tells the engine the following:

- Value is limited to between 25 and 100.
- The float value uses a step of 0.01 (this limits the fidelity of values).
- The string "Damage threshold" will be displayed in the Lumberyard Editor as a tool tip.

Editor Table

The Editor table provides additional configuration information to Lumberyard Editor on how to handle instances of the entity.

Entity class editor variables

Variable	Description
Model	CGF model to be rendered over an entity instance.
ShowBounds	Flag indicating whether or not a bounding box is drawn around an entity instance when selected.
AbsoluteRadius	
Icon	BMP icon to be drawn over an entity instance.
IconOnTop	Flag indicating whether or not the icon is drawn over or under an entity instance.
DisplayArrow	
Links	

Functions

A script entity can include several callback functions called by the engine or game system. See Entity System Script Callbacks (p. 120) for more information.

Using Entity State

The Entity system provides a simple state-switching mechanism for script entities.

Each state consists of the following:

- Name (string)
- Lua table within the entity table, identified with the state name
- **OnEndState()** function (optional)
- **OnBeginState()** function (optional)
- Additional callback functions (optional) (See Entity System Script Callbacks (p. 120))

To declare states for an entity:

All entity states must be declared in the entity's main table to make the Entity system aware of them. The following examples show how to declare "Opened", "Closed", and "Destroyed" states.

```
AdvancedDoor =
{
    Client = {},
    Server = {},
    PropertiesInstance = ...
    Properties = ...
    States = {"Opened","Closed","Destroyed"},
}
```

To define an entity state:

Entity states can be either on the server or client (or both). The definition for a server-side "Opened" state might look as follows:

```
AdvancedDoor.Server.Opened =
{
 OnBeginState = function( self )
  if(self.Properties.bUsePortal==1)then
   System.ActivatePortal(self:GetWorldPos(),1,self.id);
  end;
  self.bUpdate=1;
  self.lasttime=0;
  AI.ModifySmartObjectStates( self.id, "Open-Closed" );
  self:Play(1);
  end,

 OnUpdate = function(self, dt)
  self:OnUpdate();
 end,
}
```

To set an entity's initial state:

Initially, an entity has no state. To set an entity's state, use one of the entity's callback functions (not to be confused with an entity state's callback function) to call its `GotoState()` method, shown in the following example. Once the entity state is set, the entity resides in that state and events will also be directed to that state.

```
function AdvancedDoor:OnReset()
    self:GotoState("Opened");
end
```

To change an entity's state:

Transitioning from the current state to any other state can also be done using the `GotoState()` method, as follows.

```
function AdvancedDoor.Server:OnHit(hit)
    ...
    if(self:IsDead())then
        self:GotoState("Destroyed");
    end
end
```

To query an entity's state:

Querying the state the entity is currently in can be done using the **GetState()** method, as follows.

```
if (self:GetState()=="Opened") then ...

if (self:GetState()~="Opened") then ...
```

Using Entity Slots

Each entity can have slots that are used to hold different resources available in Lumberyard. This topic describes how to work with entity slots.

Allocating a Slot

The following table lists the resources that can be allocated in a slot, along with the ScriptBind function used to allocate it.

Lumberyard resource	Function
static geometry	`LoadObject()` or `LoadSubObject()`
animated character	`LoadCharacter()`
particle emitter	`LoadParticleEffect()`
light	`LoadLight()`
cloud	`LoadCloud()`
fog	`LoadFogVolume()`
volume	`LoadVolumeObject()`

Modifying Slot Parameters

Each of these resource may be moved, rotated, or scaled relative to the entity itself.

- `SetSlotPos()`
- `GetSlotPos()`
- `SetSlotAngles()`
- `GetSlotAngles()`
- `SetSlotScale()`
- `GetSlotScale()`

You can add a parenting link between the slots, making it possible to have related positions.

- `SetParentSlot()`
- `GetParentSlot()`

Slot Management

To determine whether or not a specified slot is allocated, call the function `!IsSlotValid()`.

To free one slot, call `!FreeSlot()`

To free all allocated slots within the entity, call `!FreeAllSlots()`.

Loading a Slot

The following example illustrates loading a slot in a script function.

```
local pos={x=0,y=0,z=0};
self:LoadObject(0,props.fileModel);
self:SetSlotPos(0,pos);
self:SetCurrentSlot(0);
```

Linking Entities

In Lumberyard Editor, you can link an entity to other entities. These links are organized inside the Entity system. Each entity can link to multiple entities. Each link has a name associated to it. See the Amazon Lumberyard User Guide for more information about grouping and linking objects.

The following example Lua script searches the Entity system for any links to other entities that are named "Generator".

```
function RadarBase:IsPowered()
 local i=0;
 local link = self:GetLinkTarget("Generator", i);

 while (link) do
  Log("Generator %s", link:GetName());

  if (link:GetState() == "PowerOn") then
   if (link.PowerConnect) then
    link:PowerConnect(self.id);
    return true;
   end
  end

  i=i+1;
  link=self:GetLinkTarget("Generator", i);
 end

 return false;
end
```

The following functions are used to read or create entity links:

- `CountLinks`
- `CreateLink`
- `GetLink`
- `GetLinkName`
- `GetLinkTarget`
- `RemoveAllLinks`
- `RemoveLink`
- `SetLinkTarget`

Exposing an Entity to the Network

A script entity can be a serialized value on the network. This approach is done by setting the values on the server and having them automatically synchronized on all the clients. It also makes it possible to invoke client/server RMI functions.

Keep in mind the following limitations:

- There is no notification when a serialized value has changed.
- Values are controlled on the server only, there is no way to set values on the client.

Exposing a Script Entity to CryNetwork

To define the network features of an entity, call the ScriptBind function `Net.Expose()`, as illustrated in the following code. This code is written inside a Lua script within the global space, rather than in a function.

```
Net.Expose {
 Class = DeathMatch,
 ClientMethods = {
  ClVictory              = { RELIABLE_ORDERED, POST_ATTACH, ENTITYID, },
  ClNoWinner             = { RELIABLE_ORDERED, POST_ATTACH, },

  ClClientConnect        = { RELIABLE_UNORDERED, POST_ATTACH, STRING, BOOL },
  ClClientDisconnect     = { RELIABLE_UNORDERED, POST_ATTACH, STRING, },
  ClClientEnteredGame    = { RELIABLE_UNORDERED, POST_ATTACH, STRING, },
 },
 ServerMethods = {
  RequestRevive          = { RELIABLE_UNORDERED, POST_ATTACH, ENTITYID, },
  RequestSpectatorTarget = { RELIABLE_UNORDERED, POST_ATTACH, ENTITYID, INT8
},
 },
 ServerProperties = {
  busy  = BOOL,
 },
};
```

RMI functions

The RMI function is defined in either the ClientMethods and ServerMethods tables passed to the `Net.Expose()` function.

Order flags:

- UNRELIABLE_ORDERED
- RELIABLE_ORDERED
- RELIABLE_UNORDERED

The following descriptors control how the RMI is scheduled within the data serialization.

RMI attach flag	Description
NO_ATTACH	No special control (preferred)
PRE_ATTACH	Call occurs before data serialized
POST_ATTACH	Call occurs after the data serialized

The following example shows a function declaration:

```
function DeathMatch.Client:ClClientConnect(name, reconnect)
```

The following examples illustrate a function call:

```
self.allClients:ClVictory( winningPlayerId );
```

```
self.otherClients:ClClientConnect( channelId, player:GetName(), reconnect );
```

```
self.onClient:ClClientConnect( channelId, player:GetName(), reconnect );
```

See RMI Functions (p. 341) for more details.

Note
Note: Script networking doesn't have an equivalent to the dependent object RMIs.

ServerProperties table

The entity table also contains a ServerProperties table that indicates which properties need to be synchronized. This is also the place to define the variable type of the value.

Exposing a Script Entity to CryAction

In addition, you must create a game object in CryAction and bind the new game object to the network game session. The following example shows the code placed in the `OnSpawn()` function:

```
CryAction.CreateGameObjectForEntity(self.id);
CryAction.BindGameObjectToNetwork(self.id);
```

You can also instruct the game object to receive a per-frame update callback, as in the following function call to CryAction:

```
CryAction.ForceGameObjectUpdate(self.id, true);
```

The script entity receive the `OnUpdate()` function callback of its Server table.

```
function Door.Server:OnUpdate(frameTime)
-- some code
end
```

Note
Adding update callback code to your script entity can decrease the performance of a game.

Event Bus (EBus)

The event bus (EBus) ia a general-purpose system for dispatching messages. EBuses support many different uses depending how one configures and uses a given EBus. Such uses include, but aren't limited to the following:

- **Abstraction** – Minimize hard dependencies between systems.
- **Event-driven programming** – Eliminate polling patterns for more scalable and performant software.
- **Cleaner application code** – Safely dispatch messages without concern for who is listening or whether anyone at all is listening.
- **Concurrency** – Queue events from various threads for safe execution on another or for distributed system applications.
- **Predictability** – Support the ordering of listeners on a given bus.
- **Debugging** – Intercept messages for reporting, profiling, and introspection purposes.

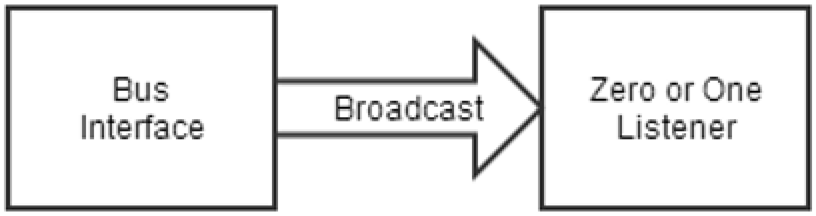
Bus Configurations

EBuses can be configured for various usage patterns.

Topics

Single Handler

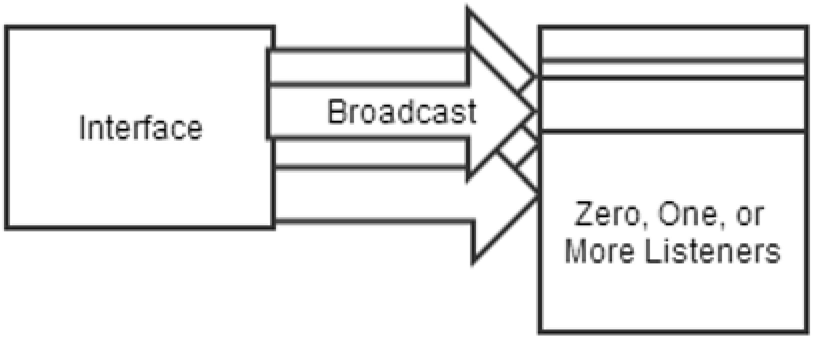
The simplest configuration is a many-to-one (or zero) communication bus, much like a singleton pattern. There is at most one listener, to which anyone can dispatch events. Senders need not manually check and dereference pointers. If no system is listening on the bus, the event is simply ignored.



```
static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_SINGLE;
// There's only one listener (Single Handler case)
static const AZ::EBusContainerTypes BusContainer = AZ::EBCT_SINGLE;
// There is only one bus, no addressing/ID
```

Many Handlers and Listeners

Another common configuration is one in which many listeners may be present. This configuration is used to implement observer patterns, subscribing to system events, or for general-purpose broadcasting.



Without ordering

```
static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_MULTI;
// We allow any number of listeners (without defined order).
static const AZ::EBusContainerTypes BusContainer = AZ::EBCT_SINGLE;
// There is only one bus, no addressing/ID.
```

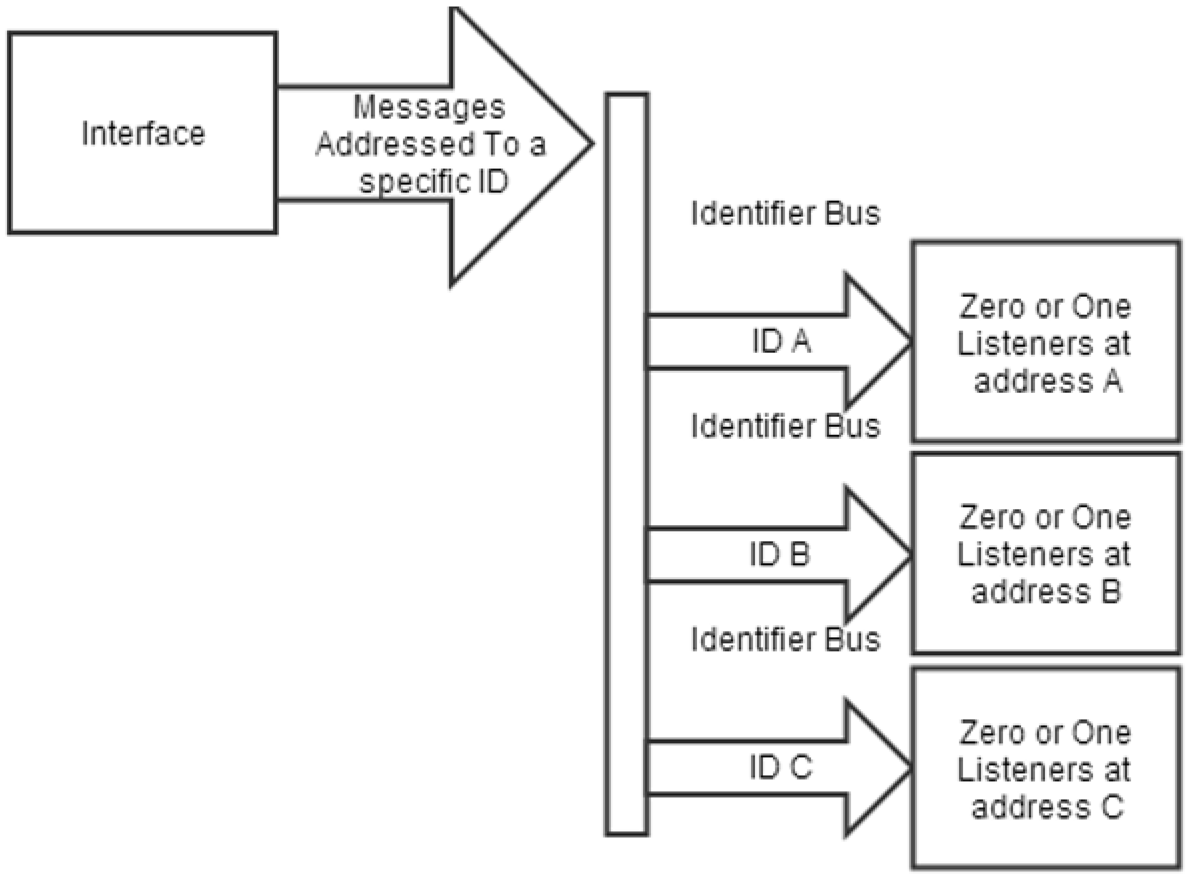
With ordering

```
static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_MULTI_ORD;
// We allow any number of listeners (without defined order).

static const AZ::EBusContainerTypes BusContainer = AZ::EBCT_SINGLE;
// There is only one bus, no addressing/ID.
```

```
struct BusListenerOrderCompare : public AZStd::binary_function<MyBusInterface*,
 MyBusInterface*, bool>

// implement custom Listener/Event ordering function
{
AZ_FORCE_INLINE bool operator()(const MyBusInterface* left, const MyBusInterface*
 right) const { return left->GetOrder() < right->GetOrder();}
};
```

Single Handler with Identified Bus

EBuses also support addressing on a custom ID, with effectively a separate bus for each ID but using a common interface. This allows one to broadcast messages to all listeners or choose to broadcast to a listener for a specific ID. A common use case is in components, where communication between the various components of a single entity is required, or with components on a separate but related entity. In this case the entity ID is used as the address.


Interface
Messages Addressed To a specific ID
Identifier Bus
ID A
Zero or One Listeners at address A
Identifier Bus
ID B
Zero or One Listeners at address B
Identifier Bus
ID C
Zero or One Listeners at address C

Without ordering channels

```
static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_SINGLE;
```

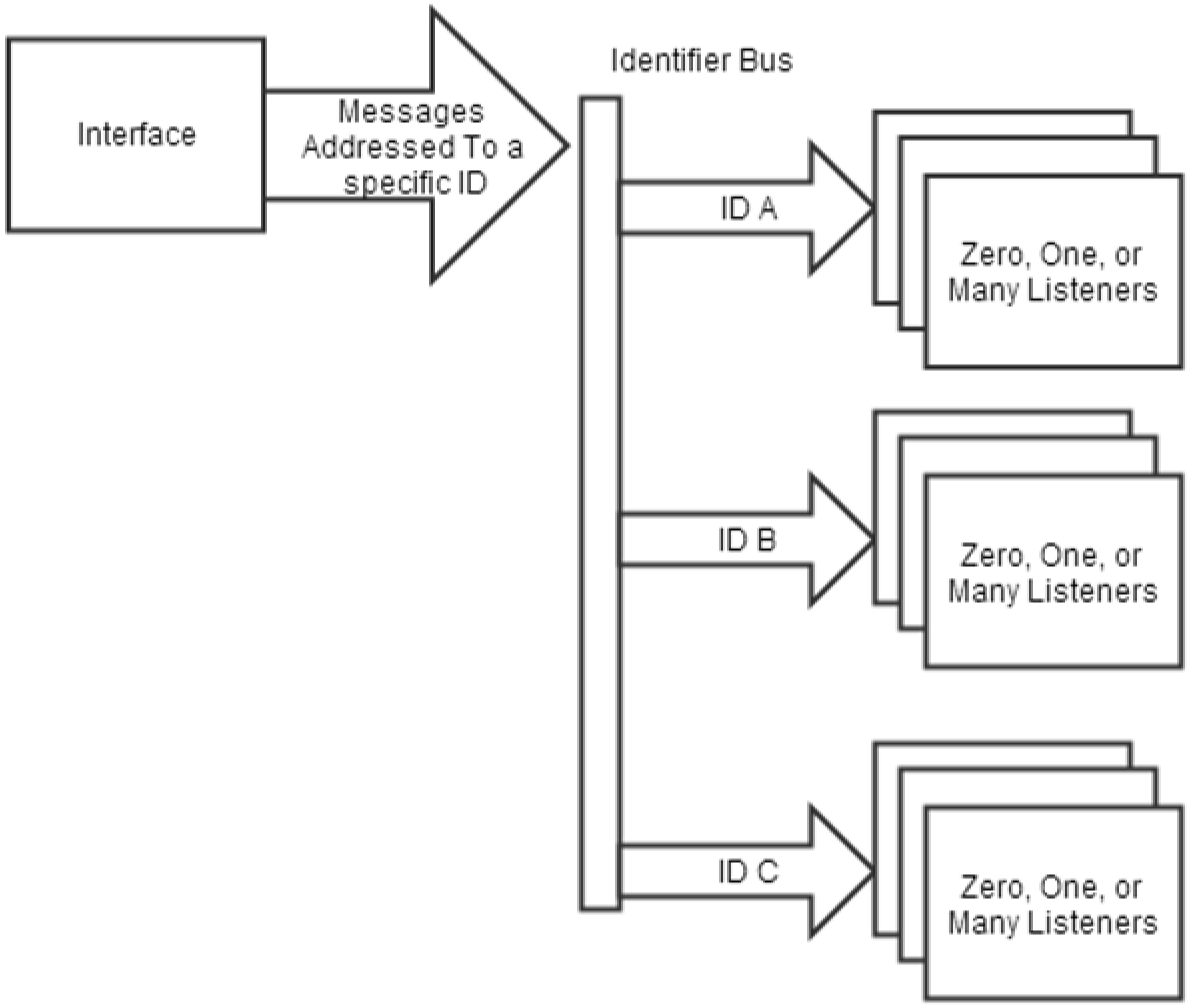
```
// We allow only a single handler for each ID.

static const EBusContainerTypes BusContainer = AZ::EBCT_ID_UNORDERED;
// We address on an ID, but without ordering.

using BusIdType = EntityId;
```

With ordering channels

```

static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_SINGLE;
// We allow only a single handler for each ID.

static const EBusContainerTypes BusContainer = AZ::EBCT_ID_ORDERED;
// We address on an ID, but without ordering.

using BusIdType = EntityId;
// We address the bus EntityId.

using BusIdOrderCompare = AZStd::greater<BusIdType>;
// Order by Entity Id.
```

Many Handlers/Listeners with Identified Bus

Note that in the above configuration, only one listener is allowed per address. This is often desirable to enforce ownership of a given bus for a given ID, similar to the singleton case above. In the event that you want to allow any number of listeners on a given address, you can configure the bus accordingly.


Interface
Messages Addressed To a specific ID
Identifier Bus
ID A
ID B
ID C
Zero, One, or Many Listeners
Zero, One, or Many Listeners
Zero, One, or Many Listeners

Without ordering channels

```
static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_MULTI;
// We allow any number of listeners on each ID.

static const EBusContainerTypes BusContainer = AZ::EBCT_ID_UNORDERED;
// We address on an ID, but without ordering.

using BusIdType = EntityId;
```

With ordering channels

```
static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_MULTI;
// We allow any number of listeners on each ID.

static const EBusContainerTypes BusContainer = AZ::EBCT_ID_ORDERED;
```

```
// We address on an ID, but with ordering.

using BusIdType = EntityId;
// We address the bus EntityId.

using BusIdOrderCompare = AZStd::greater<BusIdType>;
// Order by Entity Id.
```

Synchronous vs. Asynchronous

EBus supports both synchronous and asynchronous (queued) messaging. Invoking an EBus event synchronously results in immediate dispatch to any and all listeners. While this does limit opportunities for asynchronous programming, it offers benefits in some scenarios:

- Synchronous messages don't require storing a closure. Arguments are forwarded directly to callers.
- Synchronous messages allow you to retrieve an immediate result from a listener (event return value).
- Synchronous messages have no latency.

However, using asynchronous messaging patterns offers the following possibilities:

- Asynchronous messages are more future proof and create more opportunities for parallelism
- Asynchronous messages support queuing messages from any thread, dispatching them on a safe thread, such as the main thread, or any other thread.
- Code written using asynchronous messages is already tolerant to latency and is easily migrated to actor models and other distributed platforms.
- Performance of the code firing off the events isn't reliant on the cost of listener event handlers.
- Asynchronous messages can improve i-cache and d-cache performance due to fewer virtual function calls. This is important in performance-critical code.

Additional Features

EBuses contain other features to address various patterns and use cases:

- Ability to cache a bus pointer to which messages can be dispatched. This is handy for identified (ID) buses. Rather than looking up the bus instance by ID on each event, cache once for faster dispatching.
- Ability to queue any callable on a bus. When using queued messaging, one can queue a lambda or bound function against a bus for execution on another thread. Very useful for general purpose thread-safe queuing.

Usage and Example

Declaring an EBus is much like declaring any virtual interface class in C++. However, various configuration options can be specified to control at compile-time how the EBus is generated and how it behaves.

Here is a simple example of a basic interface and associated EBus.

```

class ExampleInterface : public AZ::EBusTraits
```

```
{
public:
// ------------------ BUS CONFIGURATION ---------------
// These override the defaults in EBusTraits.

static const AZ::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_SINGLE;
// There's only one listener (Single Handler case).

static const AZ::EBusContainerTypes BusContainer = AZ::EBCT_SINGLE;  // There
is only one bus, no addressing/ID
// -------------------- OTHER ------------------------

~ExampleInterface() override { };
// --------------------- Listener interface -------------
// Listeners inherit from ExampleInterface::Bus::Listener

virtual void DoSomething() = 0;
// Listeners are required to implement this because it's pure virtual.

virtual void SomeMessage() { }
// Listeners can override this, but are not required to.

virtual bool ReturnedValue(int x) = 0;
// Returns a value and has a paramter.};

using ExampleInterfaceBus = AZ::EBus<ExampleInterface>;
```

The bus configuration options are key to controlling how the bus behaves with respect to the different modes discussed previously.

```
static const DH::EBusEventGroupContainerTypes EventGroupContainer =
AZ::EBBCT_SINGLE;
// There's only one listener (Single Handler case).
```

The `EventGroupContainer` trait determines the type of container used internally to store listeners. In this case we're specifying that we want to allow only a single listener/handler, such as the Single Handler configuration. Practical uses include the following:

- You have a singleton pattern where various systems need to post messages or requests to a single system elsewhere in the codebase.
- You have a component that owns a given bus for the entity. That is, mesh component owns mesh-related queries for the entity it owns.

```


static const DH::EBusContainerTypes BusContainer = AZ::EBCT_SINGLE;
// There is only one bus, no addressing/ID.
```

The `BusContainer` trait determines how individual communication channels for the bus are stored. In other words, only a single bus exists with no ID or addressing requirements.

A practical use cases includes any global bus that isn't tied to a specific entity or other application-specific ID or object.

EventGroupContainer Configuration Options

- **EBBCT_SINGLE** – There is only one listener/handler for this bus, such as the Single Handler or Single Handler with Identified Bus options.
- **EBBCT_MULTI** – There can be any number of listeners for this bus, and ordering is not a concern. This includes the Many Handlers/Listeners or Many Handlers/Listeners with Identified Bus options.
- **EBBCT_MULTI_ORD** – There can be any number of listeners, and we want to always notify listeners in a particular order. The `BusListenerOrderCompare` definition allows for arbitrary customization of ordering.

BusContainer Configuration Options

- **EBCT_SINGLE** – There is only one instance of this bus, with no ID or addressing. This includes the Single Handler or Many Handlers/Listeners options.
- **EBCT_ID_UNORDERED** – There can be any number of channels, based on a customizable ID type. We're not concerned with the order in which channels are notified when sending events on the bus without specifying an ID.
- **EBCT_ID_ORDERED** – There can be any number of channels based on a customizable ID type. However, when sending events on the bus without specifying an ID, we want to control the order in which individual bus channels are notified. The `BusIdOrderCompare` definition allows for arbitrary customization of ordering.

Implementing a Listener

A listener of a bus derives from `DH::EBus<x>::Listener`. Because we convenience typedef this as `Bus` in the above example, this means that we can derive from `Bus::Listener`.

```
class MyListener : protected ExampleInterface::Bus::Listener // note - not de
rived from ExampleInterface, but instead the bus listener.
{
public:
AZ_CLASS_ALLOCATOR(MyListener, AZ::SystemAllocator, 0) // enable dhnew.

void Activate();

protected:
// Implement the listener interface:
virtual void DoSomething() override;
// note:   Override specified.
virtual void SomeMessage() override;
virtual bool ReturnedValue(int x) override;
};
```

Note that listeners are not automatically connected to a bus but are automatically disconnected since the destructor of `Listener` calls `BusDisconnect`.

In order to actually connect to the bus and start listening, a `Listener` must call `BusConnect()`:

```
 void MyListener::Activate()
 {

 ExampleInterface::Bus::Listener::BusConnect();
// this could also be just BusConnect, but if you're on multiple busses, you
must specify which.
}
```

`BusConnect()` can be called at any time from any thread.

If your bus is addressed, connect to the bus for a specific ID by passing the ID to `BusConnect()`. To listen on all addresses, call `BusConnect()` without an ID.

```
Example::Bus::Listener::BusConnect(5); // connect to the bus at address 5.
```

Sending Messages to a Bus

Anyone who can include the header can send messages to the bus at any time. Using the above example, a completely unrelated class can issue `DoSomething` on that bus:

```

#include "ExampleInterface.h"
// note:  We don't need to include MyListener.h
...
...
EBUS_EVENT(ExampleInterface::Bus, DoSomething);
EBUS_EVENT(ExampleInterface::Bus, ReturnedValue, 5);
// calls the bus without reading the result, packs 5 as the first parameter.
```

If your bus is addressed, events can be sent to a specific ID or globally for all IDs.

```

EBUS_EVENT(Example::Bus, Test);
// broadcasts to EVERYONE regardless of address (even if the bus has ad
dresses)EBUS_EVENT_ID(5, Example::Bus, Test).
// broadcasts only to any listener connected to address 5.
```

Retrieving Return Values

If you make a synchronous call (EBUS_EVENT) you can also supply a variable to place the result in.

```

bool result = false;
// ALWAYS INITIALIZE YOUR RESULT - since there may be nobody listening to the
bus, your result may not be populated.

EBUS_EVENT_RESULT(result, ExampleInterface::Bus, ReturnedValue, 2);
```

In the above example, if nobody is listening, the result is not be modified. If someone is listening, `operator=()` will be called on the given variable (in this case, `'result'`) for each responder.

Return Values on Multiple Buses

Sometimes you need to aggregate the return value of a function where there might be multiple listeners. For example, suppose you wanted to send a message out to all listeners asking whether anyone objected to shutting the application down. If anyone returns true, you want to abort shutdown. The following would not suffice:

```
bool returnValue = false;
EBUS_EVENT_RESULT(returnValue, SomeInterface::Bus, DoesAnyoneObject);
```

Because the bus issues `operator=` on each returned listener, `returnValue` would contain only the final result of that final listener.

Instead, you can create a class that overrides `operator=` to collect your results. There are several built-in types for this, and of course, you can make your own:

```

#include <dhcore/ebus/results.h>

...
DH::EBusAggregateResults<bool> results;
EBUS_EVENT_RESULT(results, SomeInterface::Bus, DoesAnyoneObject);

// results now contains a vector of all results from all responders.

// alternative:
DH::EBusLogicalResult<bool, DHStd::logical_or<bool> > response(false);
EBUS_EVENT_RESULT(response, SomeInterface::Bus, DoesAnyoneObject);

// response now contains each result, using logical or operation.  So all re
sponses are OR'd with each other.
```

Inside `results.h` you'll find a few more building blocks - arithmetic results for example.

Asynchronous/Queued Buses

To declare a bus such that events can be queued, add this to the bus declaration:

```
static const bool EnableEventQueue = true;
```

EBUS_QUEUE_EVENT (and variants) can now be used to queue events on a bus or flushing later from a controlled location or thread.

To flush the queue at the appropriate location or thread, do the following:

```
ExampleInterface::Bus::ExecuteQueuedEvents();
```

File Access

This section covers tools available for tracking and accessing game files.

This section includes the following topics:

CryPak File Archives

The CryPak module enables you to store game content files in a compressed or uncompressed archive.

Features

- Compatible with the standard zip format.
- Supports storing files in an archive or in the standard file system.
- Data can be read in a synchronous and asynchronous way through `IStreamCallback` (max 4GB offset, 4GB files).
- Files can be stored in compressed or uncompressed form.
- Uncompressed files can be read partially if required.
- File name comparison is not case sensitive.
- Supports loading of `.zip` or `.pak` files up to 4GB in size.

Unicode and Absolute Path Handling

Internally, all path-handling code is ASCII-based; as such, no Unicode (16-bit characters for different languages) functions can be used—this is to save memory and for simplicity. Because games can and should be developed with ASCII path names, no real need for Unicode exists. Game productions that don't follow these requirements have issues integrating other languages. For example, because a user might install a game to a folder with Unicode characters, absolute path names are explicitly avoided throughout the whole engine.

Layering

Usually the game content data is organized in several `.pak` files, which are located in the game directory. When a file is requested for an opening operation, the CryPak system loops through all registered `.pak` files. `.pak` files are searched in order of creation. This allows patch `.pak` files, which have been added to the build later, to be in a preferred position. It is also possible to mix `.pak` files with loose files, which are stored directly in the file system (not in a `.pak` file). If a file exists as a loose file as well as in a `.pak` archive, the loose file is preferred when the game is in **devmode**. However, to discourage cheating in the shipped game, the file stored in the `.pak` is preferred over the loose file when the game is not run in devmode.

Slashes

Usually forward slashes (/) are used for internal processing, but users may enter paths that contain backslashes.

Special Folder Handling

You can use the path alias `%USER%` to specify a path relative to the user folder. This might be needed to store user-specific data. Windows can have restrictions on where the user can store files. For example, the program folder might not be writable at all. For that reason, screenshots, game data, and other files should be stored in the user folder. The following are examples of valid file names and paths:

```
%USER%/ProfilesSingle/Lisa.dat
game/Fred.dat
```

Internals

- A known implementation flaw exists where using more than approximately 1000 files per directory causes problems.
- Format properties:
  - The `.zip` file format stores each file with a small header that includes its path and filename in uncompressed text form. For faster file access, a directory is listed at the end of the file. The directory also stores the path and filename in uncompressed text form (redundant).

Creating a pak file using 7-Zip

To create a `.pak` file with 7-Zip's `7za.exe` command line tool, use the following syntax:

```
7za a -tzip -r -mx0 PakFileName [file1 file2 file3 ...] [dir1 dir2 ...]
```

Dealing with Large Pak Files

The zip RFC specifies two types of `.zip` files, indicated by `.zip` format version 45. Old `.zip` files can have a 4GB offset, but if legacy I/O functions are used, it is only possible to seek +- 2GB, which becomes the practical limit. The 4GB offsets have nothing to do with native machine types and do not change size across platforms and compilers, or configurations. The offsets for older versions of `.zip` files are in a machine independent `uint32`; the offsets for the new version `.zip` files are in `uint64`, appended to the old version `structs`. The version a `.zip` file uses is located in the header of the `.zip` file. Applications are free to not support the newer version. For more information, see the .ZIP File Format Specification.

Manual splits are not necessary, as RC supports auto-splitting:

- `zip_sizesplit` – Split `.zip` files automatically when the maximum configured or supported compressed size has been reached. The default limit is 2GB.
- `zip_maxsize` – Maximum compressed size of the `.zip` file in kilobytes (this gives an explicit limit).

Splitting works in all cases and supports multi-threading and incremental updates. It expands and shrinks the chain of necessary zip-parts automatically. Sorting is honored as much as possible, even in face of incremental modifications, but individual files can be appended to the end of the parts to fill in the leftover space even if this violates the sort order.

For more information about zip files, see Zip File Format Reference by Phil Katz.

Accessing Files with CryPak

In this tutorial you will learn how file reading and writing works through `CryPak`. The tutorial teaches you how to add new files to your project, read files from the file system and from pak archives, and write files to the file system.

Topics

Preparation

This tutorial demonstrates two different methods of loading a file: from inside a `.pak` archive, and directly from the file system. Before you can start, you need a file in a `.pak` archive, and a file with the same name (but with different content) in the file system. To verify which file is loaded, the example makes use of the content inside each text file.

To prepare sample files

1. Create a text file named `ExampleText.txt`.
2. Using a text editor, open `ExampleText.txt` and type in the following text:
3.
```
This sample was read from the .pak archive
```
4. Save the file.
5. Inside the `GameSDK` folder, create a subfolder called `Examples`.
6. Add the `ExampleText.txt` file to the `Examples` folder so that the path looks like this:

   `<root>\GameSDK\Examples\ExampleText.txt`
7. Run the following command from the directory *`root`*`\GameSDK`:

```
..\Tools\7za.exe a -tzip -r -mx0 Examples.pak Examples
```

   This command uses the executable file `7za.exe` (located in the `Tools` folder) to create an archive of the `Examples` folder called `Examples.pak`. Because you ran the command from the `GameSDK` folder, the archive was saved to the GameSDK folder. The .pak file contains only the file Examples\ExampleText.txt.

8. Using a text editor, change the text inside the *<root>*\GameSDK\Examples\ExampleText.txt file to something different, for example:

```
This sample was read from the file system
```

Now you have two different text files with the same destination path, except that one is stored directly in the file system, and the other is inside the .pak file.

Reading Files with CryPak

Now you can write some code to read the information from the ExampleText.txt file that you created.

1. Type the following, which contains the if-else statement that frames the code. The ReadFromExampleFile() function will read the contents of the file and return true if it succeeds, and false if not.

```
    char* fileContent = NULL;
    if (!ReadFromExampleFile(&fileContent))
    {
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "Read
FromExampleFile() failed");
    }
    else
    {
        CryLogAlways("ExampleText contains %s", fileContent);
        [...] // this line will be added later on
    }
```

   If ReadFromExampleFile() is successful in reading ExampleText.txt, fileContent will be the space in memory that contains the text that it read.

2. Type the following, which stubs out the ReadFromExampleFile() function.

```
bool ReadFromExampleFile(char** fileContent)
{
    CCryFile file;
    size_t fileSize = 0;
    const char* filename = "examples/exampletext.txt";

    [...]
}
```

   - file of type CCryFile can make use of CryPak to access files directly from the file system or from inside a .pak archive.
   - fileSize - Defines the end of the message. In this case, reading does not end by detecting the null character '\0'.
   - filename - Specifies the path of the file to be loaded and is case-insensitive.

3. Type the following, which uses CryPak to search the file.

```
    char str[1024];
    if (!file.Open(filename, "r"))
    {
        sprintf(str, "Can't open file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }
```

- `Open()` invokes `CryPak` to search the file specified by `filename`.
- File access mode `"r"` specifies that a plain text file is going to be read. To read a binary file, use `"rb"` instead.

4. Type the following, which gets the length of the file. If the file is not empty, it the allocates the memory required as indicated by the file length. It then reads the file content. It aborts if the size of the content is not equal to the file length.

```
    fileSize = file.GetLength();
    if (fileSize <= 0)
    {
        sprintf(str, "File is empty, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    char* content = new char[fileSize + 1];
    content[fileSize] = '\0';

    if (file.ReadRaw(content, fileSize) != fileSize)
    {
        delete[] content;
        sprintf(str, "Can't read file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }
```

- `content` is the local pointer to a `char` array in memory which gets initialized by the length returned by `GetLength()` and an extra null character.
- `ReadRaw` fills `content` with the information read from the text file. In case of a failure, the allocated memory of `content` is freed.

5. Type the following, which closes the file handle and sets the the `fileContent` pointer so that the locally created data can be used outside the function. Finally, it returns `true` since the reading was successful.

```
    file.Close();

    *fileContent = content;
    return true;
```

Note
In the example, the caller of `ReadFromExampleFile()` is responsible for freeing the heap memory which has been allocated to store the data from the text file. Thus, after the data has been used, be sure to add the call `delete[] fileContent;`.

6. To check if the reading was successful, run the game and check the `Game.log` file.

Complete example code (file reading)

`Calling ReadFromExampleFile()`

```
    char* fileContent = NULL;
    if (!ReadFromExampleFile(&fileContent))
    {
       CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "ReadFromExample
File() failed");
    }
    else
    {
        CryLogAlways("ExampleText contains %s", fileContent);
        delete[] fileContent;
    }
```

`ReadFromExampleFile() implementation`

```
bool ReadFromExampleFile(char** fileContent)
{
    CCryFile file;
    size_t fileSize = 0;
    const char* filename = "examples/exampletext.txt";

    char str[1024];
    if (!file.Open(filename, "r"))
    {
        sprintf(str, "Can't open file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    fileSize = file.GetLength();
    if (fileSize <= 0)
    {
        sprintf(str, "File is empty, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    char* content = new char[fileSize + 1];
    content[fileSize] = '\0';

    if (file.ReadRaw(content, fileSize) != fileSize)
    {
        delete[] content;
        sprintf(str, "Can't read file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    file.Close();

    *fileContent = content;
    return true;
}
```

Writing to File System Files With CryPak

Writing a file is similar to the process for reading one. To write to files, you use `CCryFile::Write`, which always writes to the file system and never to `.pak` archives. For information on writing files to archive files, see Modifying Paks With CryArchive (p. 297).

1. Type the following, which contains the `if-else` statement that frames the code for writing to a file. The `WriteToExampleFile()` function write will write the contents of the file and return `true` if it succeeds, and `false` if not.

```
    char* newContent = "File has been modified";
    bool appendToFile = false;
    if (!WriteToExampleFile(newContent, strlen(newContent), appendToFile))
    {
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "WriteTo
ExampleFile() failed");
    }
    else
    {
        CryLogAlways("Text has been written to file, %s", newContent);
    }
```

- `WriteToExampleFile()` takes the following three parameters:
  - `newContent` - The text which will be written to `ExampleText.txt` on the file system.
  - `strlen(newContent)` - Returns size of `newContent`, which is the number of bytes to be written.
  - `appendToFile` - `true` if `newContent` will be added to the already existing content; `false` if the file will be overwritten.

2. Type the following for the `WriteToExampleFile)` function.

```
bool WriteToExampleFile(char* text, int bytes, bool appendToFile)
{
    CCryFile file;
    const char* filename = "examples/exampletext.txt";

    assert(bytes > 0);
    char* mode = NULL;
    if (appendToFile)
        mode = "a";
    else
        mode = "w";

    char str[1024];
    if (!file.Open(filename, mode))
    {
        sprintf(str, "Can't open file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }
```

```
 [...]

    file.Close();
    return true;
}
```

- `mode` specifies if the text is to be appended to the existing file or if it will overwrite existing file contents. `"w"` means 'write' to a clean file, and `"a"` means 'append' to the existing file.

3. The final step writes the text to the file and returns the number of bytes written, or an error message if none were written.

```
    int bytesWritten = file.Write(text, bytes);
    assert(bytesWritten == bytes);

    if (bytesWritten == 0)
    {
        sprintf(str, "Can't write to file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }
```

- `bytesWritten` tells how many bytes were written by calling the `Write()` function.

Modifying Paks With CryArchive

This section contains a short example that shows how files are added, updated and removed from an archive. The example intentionally uses the `USER` folder instead of the `GameSDK` folder because the `.pak` files inside the `GameSDK` folder are loaded by default at startup and therefore are marked as `Read-Only`. (Files in the `USER` folder are not loaded by default at startup.)

```
    string pakFilename = PathUtil::AddSlash("%USER%") + "Examples.pak";
    const char* filename = "Examples/ExampleText.txt";
    char* text = "File has been modified by CryArchive";
    unsigned length = strlen(text);

    _smart_ptr<ICryArchive> pCryArchive = gEnv->pCryPak->OpenArchive(pakFile
name.c_str(), ICryArchive::FLAGS_RELATIVE_PATHS_ONLY | ICryArchive::FLAGS_CRE
ATE_NEW);
    if (pCryArchive)
    {
        pCryArchive->UpdateFile(filename, text, length, ICryArchive::METH
OD_STORE, 0);
    }
```

- `UpdateFile()` - Modifies an existing file inside the `.pak` archive or creates a new one if it does not exist.
- `ICryArchive::FLAGS_CREATE_NEW` - Forces a new `.pak` file to be created. If you want to add (append) files, remove this flag.

- To remove files or folders from an archive, use one of the following commands in place of `UpdateFile()`: `RemoveFile()`, `RemoveDir()` or `RemoveAll()`.

CryPak Details

Initialization

To ensure that `.pak` files can be accessed from game code at anytime, the `CrySystem` module initializes `CryPak` in `CSystem::Init` by calling the following functions:

- `InitFileSystem(startupParams.pGameStartup);`
- `InitFileSystem_LoadEngineFolders();`

Tip
A good spot to test game initialization is in inside `Game.cpp` at the beginning of `CGame::Init`.

Pak file type priorities

Whether CryPak processes files in the file system first, or files in `.pak` files first, depends on the value of `pakPriority`. The default value of `pakPriority` depends on the configuration settings of your build, but it can also manually be changed by assigning the cvar `sys_PakPriority` the values `0, 1, 2 or` 3. The meaning of these values is show in the enum `EPakPriority`:

`PakVars.h`

```
enum EPakPriority
{
    ePakPriorityFileFirst = 0,
    ePakPriorityPakFirst  = 1,
    ePakPriorityPakOnly   = 2,
    ePakPriorityFileFirstModsOnly = 3,
};
```

Pak loading and search priorities

The reason for adding the new pak file to the `GameSDK` folder in this example is because `.pak` files are loaded from the `GameSDK` path first. The loading order and search order of .pak file folders are as follows. Note that the loading order and the search order are the reverse of each other.

.pak file load order

1. `GameSDK`: *`<root>`*`\GameSDK\*.pak`
2. `Engine`: *`<root>`*`\Engine\`
   a. `Engine.pak`
   b. `ShaderCache.pak`
   c. `ShaderCacheStartup.pak`
   d. `Shaders.pak`
   e. `ShadersBin.pak`
3. Mods: `root\Mods\MyMod\GameSDK\*.pak` (this assumes that you run the game with the command argument `-mod "`*`MyMod`*`"`)

.pak file search order

1. `Mods` If more than one mod folder exists, they will be checked in the reverse order in which they were added.
2. `Engine`
3. `GameSDK`

Tracking File Access

It's possible to track invalid file reads that occur during game run time. The error message `Invalid File Access` occurs when an attempt is made to read or open open files from a thread that is not the streaming thread. These file access operations can cause stalls that can be quite severe.

Note
Only access attempts from the main thread and render thread are logged. This feature is disabled in RELEASE builds.

CVars

The following cvars enable different options for tracking file access.

`sys_PakLogInvalidFileAccess`

`1` (default):

- Access is logged to `game.log`.
- Generates a `perfHUD` warning.
- The warning is displayed in red in the upper left corner of the screen.
- A 3 second-stall in non-release builds is induced.

`sys_PakMessageInvalidFileAccess`

- When a file is accessed, creates a popup dialog on the PC. At this point, you can choose to break into the debugger, or continue.

Where invalid access is defined

The points which define when a file access attempt is considered invalid are set by implementing `ICryPak::DisableRuntimeFileAccess` to return true or false. The points may need to be tweaked for single player and multiplayer games.

Exceptions

To add exceptions to file access tracking so that you can ignore files like `game.log`, create an instance of `CDebugAllowFileAccess` in the scope which accesses the file.

Resolving file access callstacks

The files that you collect with `pak_LogInvalidFileAccess 2` must have their callstacks resolved. To do this requires the following tools from the `XenonStackParse` folder of the `Tools` directory.:

- The `.pdb` files from the build
- The `XenonStackParse` tool
- The `ProcessFileAccess.py` helper script

The directory structure for running `ProcessFileAccess.py` should resemble the following:

```
<Root>
--> XenonStackParse
--> FileAccessLogs (this folder should contain the .pdb files)
------> Processed (this folder contains the output from XenonStackParse)
```

Run `ProcessFileAccess.py` from the `FileAccessLogs` directory (`XenonStackParse` uses the working directory to search for the `.pdb` files). The script creates a folder called `Processed` and a file within it that contains the resolved callstack for each of the log files.

Graphics and Rendering

Lumberyard's rendering technology starts with a modern, physically-based shading core that renders materials based on real world physical parameters (such as base color, metalicity, smoothness, and specularity), allowing you to achieve realistic results using the same physically based parameters used in the highest end film rendering pipelines.

The rendering core is supplemented by a rich set of the most frequently used real time lighting, shading, special effects, and post effects features, such as physical lights, global illumination, volumetric fog, procedural weathering, particle systems, dynamic real time shadows, motion blur, bokeh depth of field, post color correction, and more.

Lumberyard's rendering engine is tightly integrated with Lumberyard Editor, so the graphical fidelity and performance achieved in your game is what you see in the editor. Changes made in the editor are instantly reflected in the fully rendered scene, allowing for immediate feedback and rapid iteration.

The Lumberyard rendering technology is designed to take maximum advantage of today's high-end PC and console platforms, while maintaining compatibility with older hardware by scaling down graphical features and fidelity without compromising the core visual elements of your scene.

This section includes the following topics:

Render Nodes

To visualize objects in a world, Lumberyard defines the concepts of the render node and render element. Render nodes represent general objects in the 3D engine. Among other things, they are used to build a hierarchy for visibility culling, allow physics interactions (optional), and rendering.

For actual rendering, render nodes add themselves to the renderer, passing an appropriate render element that implements the actual drawing of the object. This process happens with the help of render objects, as shown in the sample code below

Creating a New Render Node

The following example creates a render node called **PrismObject**. It is derived from `IRenderNode`, defined in `Code/CryEngine/CryCommon/IEntityRenderState.h`.

1. Add the interface for `IPrismObjectRenderNode` to `CryEngine/CryCommon/IEntityRenderState.h` to make it publicly available.

```
struct IPrismRenderNode : public IRenderNode
{
  ...
};
```

2. Add a new enum to the list of already defined render nodes in `CryEngine/CryCommon/IEntityRenderState.h`.

```
enum EERType
{
  ...
  eERType_PrismObject,
  ...
};
```

3. Add PrismObjectRenderNode.h to Cry3DEngine.

```
#ifndef _PRISM_RENDERNODE_
#define _PRISM_RENDERNODE_

#pragma once

class CPrismRenderNode : public IPrismRenderNode, public Cry3DEngineBase
{
public:
  // interface IPrismRenderNode
  ...

  // interface IRenderNode
  virtual void SetMatrix(const Matrix34& mat);
  virtual EERType GetRenderNodeType();
  virtual const char* GetEntityClassName() const { return "PrismObject"; }
  virtual const char* GetName() const;
  virtual Vec3 GetPos(bool bWorldOnly = true) const;
  virtual bool Render(const SRendParams &rParam);
  virtual IPhysicalEntity* GetPhysics() const { return 0; }
  virtual void SetPhysics(IPhysicalEntity*) {}
  virtual void SetMaterial(IMaterial* pMat) { m_pMaterial = pMat; }
  virtual IMaterial* GetMaterial(Vec3* pHitPos = 0) { return m_pMaterial; }
  virtual float GetMaxViewDist();
  virtual void GetMemoryUsage(ICrySizer* pSizer);
  virtual const AABB GetBBox() const { return m_WSBBox; }
  virtual void SetBBox( const AABB& WSBBox ) { m_WSBBox = WSBBox; }

private:
  CPrismRenderNode();
private:
  ~CPrismRenderNode();

  AABB m_WSBBox;
  Matrix34 m_mat;
  _smart_ptr< IMaterial > m_pMaterial;
  CREPrismObject* m_pRE;
};

#endif // #ifndef _PRISM_RENDERNODE_
```

4. Add PrismObjectRenderNode.cpp to Cry3DEngine.

```
#include "StdAfx.h"
#include "PrismRenderNode.h"

CPrismRenderNode::CPrismRenderNode() :m_pMaterial(0)
{
                m_mat.SetIdentity();
                m_WSBBox = AABB(Vec3(-1, -1, -1), Vec3(1, 1, 1));
              m_pRE = (CREPrismObject*) GetRenderer()->EF_CreateRE(eDATA_Pris
mObject);
                m_dwRndFlags |= ERF_CASTSHADOWMAPS | ERF_HAS_CASTSHADOWMAPS;
}

CPrismRenderNode::~CPrismRenderNode()
{
                if (m_pRE)
                                m_pRE->Release(false);

                Get3DEngine()->FreeRenderNodeState(this);
}

void CPrismRenderNode::SetMatrix(const Matrix34& mat)
{
                m_mat = mat;
              m_WSBBox.SetTransformedAABB(mat, AABB(Vec3(-1, -1, -1), Vec3(1,
 1, 1)));
                Get3DEngine()->RegisterEntity(this);
}


const char* CPrismRenderNode::GetName() const
{
                return "PrismObject";
}

void CPrismRenderNode::Render(const SRendParams& rParam, const SRendering
PassInfo &passInfo)
{
  FUNCTION_PROFILER_3DENGINE;

                if(!m_pMaterial)
                                return;

               // create temp render node to submit this prism object to the
 renderer
                CRenderObject *pRO = GetRenderer()->EF_GetOb
ject_Temp(passInfo.ThreadID());                   // pointer could be cached


                if(pRO)
                {
                                // set basic render object properties
                                pRO->m_II.m_Matrix = m_mat;
                                pRO->m_ObjFlags |= FOB_TRANS_MASK;
                                pRO->m_fSort = 0;
                                pRO->m_fDistance = rParam.fDistance;

                                // transform camera into object space
                                const CCamera& cam(passInfo.GetCamera());
                                Vec3 viewerPosWS(cam.GetPosition());

                                // set render object properties
                                m_pRE->m_center = m_mat.GetTranslation();

                             SShaderItem& shaderItem(m_pMaterial->GetShader
Item(0));

                                GetRenderer()->EF_AddEf(m_pRE, shaderItem,
pRO, passInfo, EFSLIST_GENERAL, 0, SRendItemSorter(rParam.rendItemSorter));
                }
}

void CPrismRenderNode::GetMemoryUsage(ICrySizer* pSizer) const
{
                SIZER_COMPONENT_NAME(pSizer, "PrismRenderNode");
                pSizer->AddObject(this, sizeof(*this));
}

void CPrismRenderNode::OffsetPosition(const Vec3& delta)
{
                if (m_pRNTmpData) m_pRNTmpData->OffsetPosition(delta);
                m_WSBBox.Move(delta);
                m_mat.SetTranslation(m_mat.GetTranslation() + delta);
                if (m_pRE) m_pRE->m_center += delta;
}

void CPrismRenderNode::FillBBox(AABB & aabb)
{
                aabb = CPrismRenderNode::GetBBox();
}

EERType CPrismRenderNode::GetRenderNodeType()
{
                return eERType_PrismObject;
}

float CPrismRenderNode::GetMaxViewDist()
{
                return 1000.0f;
}

Vec3 CPrismRenderNode::GetPos(bool bWorldOnly) const
```

```
{
                return m_mat.GetTranslation();
}

IMaterial* CPrismRenderNode::GetMaterial(Vec3* pHitPos)
{
                return m_pMaterial;
}
```

5. To allow client code to create an instance of the new render node, extend the following function in `/Code/CryEngine/Cry3DEngine/3DEngine.cpp`

```
...
#include "PrismRenderNode.h"
...
IRenderNode * C3DEngine::CreateRenderNode(EERType type)
{
  switch (type)
  {
  ...
  case eERType_PrismObject:
  {
    IPrismRenderNode* pRenderNode = new CPrismRenderNode();
    return pRenderNode;
  }
  ...
```

TrueType Font Rendering

CryFont is used to generate font textures that are required to render text on the screen. The various features of font rendering can be seen by using the `r_DebugFontRendering` console variable.

The output is not only to test the functionality but also to document how the features can be used.

Supported Features

CryFont supports the following features:

- Font shaders – Used to configure the appearance of fonts. Multiple passes with configurable offset and color are supported to enable generation of shadows or outlines. A sample font shader is shown in the following XML example.

```
<fontshader>
  <font path="VeraMono.ttf" w="288" h="416"/>
  <effect name="default">
    <pass>
      <color r="0" g="0" b="0" a="1"/>
      <pos x="1" y="1"/>
    </pass>
  </effect>
  <effect name="console">
    <pass>
```

```
      <color r="0" g="0" b="0" a="0.5"/>
      <pos x="2" y="2"/>
    </pass>
  </effect>
</fontshader>
```

The attributes *w* and *h* of the XML font element specify the width and height of the font texture. The order of the passes in XML defines the order in which the passes are rendered. A `<pass>` element without child elements means that the pass is rendered with the default settings. The `<pos>` tag is used to offset the font, while the `<color>` tag is used to set font color and define the transparency (with the alpha channel *a*).

- Unicode – The default font used does not support all Unicode characters (to save memory), but other fonts can be used.
- TrueType fonts as source – Cached in a small texture. Common characters are pre-cached, but runtime updates are possible and supported.
- Colored text rendering
- Adjustable transparency
- Color variations within a string – Use a value of **`$0..9`** to set one of the 10 available colors. Use **`$$`** to print the $ symbol, and **`$o`** to switch off the feature.
- Returns and tabs within a string
- Text alignment
- Computation of a string's width and height – Used internally to handle center and right alignment.
- Font size variations – Bilinear filtering allows some blurring, but no mipmaps are used so this feature has limitations in minification.
- Proportional and monospace fonts
- Pixel-perfect rendering with exact texel-to-pixel mapping for best quality.

Useful Console Commands

The following console commands provide information about font rendering.

r_DebugFontRendering
Provides information on various font rendering features, useful for verifying function and documenting usage.
- 0=off
- 1=display

r_DumpFontNames
Logs a list of fonts currently loaded.

r_DumpFontTexture
Dumps the texture of a specified font to a bitmap file. You can use `r_DumpFontTexture` to get the loaded font names.

Generating Stars DAT File

The Stars DAT file contains star data that is used in sky rendering. This topic provides information you'll need if you want to modify the data in this file. It assumes you have some familiarity with generating binary files.

Star data is located in `Build\Engine\EngineAssets\Sky\stars.dat`. This data is loaded in the function `CStars::LoadData`, implemented in the file `CRESky.cpp`.

File Format

The Stars DAT file uses a simple binary format; it can be easily modified using an editing tool. The file starts with a header, followed by entries for each star. The header specifies the number of entries in the file.

All types stored in little-endian format, float32 in IEEE-754 format.

Star data provided in the SDK is based on real-world information. Typically, you can also use existing star catalogs to populate this information for you.

The file elements are as follows:

Header (12 bytes)

Name	Offset	Type	Value
Tag	0	uint32	0x52415453 (ASCII: STAR)
Version	4	uint32	0x00010001
NumStars	8	uint32	Number of star entries in the file

Entry (12 bytes)

Name	Offset	Type	Value
RightAscension	0	float32	in radians
Declination	4	float32	in radians
Red	8	uint8	star color, red channel
Green	9	uint8	star color, green channel
Blue	10	uint8	star color, blue channel
Magnitude	11	uint8	brightness, normalized range

Anti-Aliasing and Supersampling

Perceived graphics quality in a game is highly dependent on having clean and stable images. Lumberyard offers an efficient, post-processing-based, anti-aliasing solution that can be controlled in the Console using the console variable `r_AntialiasingMode`. This solution allows game developers to set the amount of anti-aliasing needed to produce graphics that fit their needs, from very sharp images to softer blurred images. Lumberyard also supports supersampling for very high-quality rendering.

Controlling Anti-Aliasing

The following table lists the currently available anti-aliasing modes available in Lumberyard using the CVar `r_AntialiasingMode`.

Mode	CVar Value	Description
No anti-aliasing	0	Disables post-processing-based anti-aliasing. Useful for debugging. Some game developers opt to use a higher resolution rather than spending system resources on anti-aliasing.
SMAA_Low (1X)	1	Enables sub-pixel morphological anti-aliasing (SMAA), which removes jaggies (staircase artifacts) on polygon edges. This mode does not address sub pixel details.
SMAA_Med (1TX)	2	Enables SMAA with basic temporal re-projection to reduce pixel crawling.
SMAA_High (2TX)	3	Enables SMAA with enhanced temporal re-projection, including matrix jittering. This mode usually provides the best image quality but can suffer from occasionally flickering edges.

The images below illustrate the range of graphics quality that can be achieved depending on the anti-alias setting used.

Controlling Supersampling

In addition to anti-aliasing, Lumberyard supports supersampling for very-high-quality rendering. Supersampling renders the scene at a higher resolution and downscales the image to obtain smooth and stable edges. Due to the high internal rendering resolution, supersampling is very performance-heavy and only suitable for games intended to be played on high-end PCs.

VR - Oculus Rift

Lumberyard support for virtual reality (VR) features with Oculus Rift is coming soon! Once released, this topic will provide a VR setup guide for your game and additional information on integrating with Oculus Rift.

Lua Scripting

This section provides reference information and help with scripting in Lua. It also covers how to use tools including the Lua debugger and XML loader.

This section includes the following topics:

Lua Scripting Reference

This section contains reference information on Lua scripting functions. Functions are organized here based on the system they are defined in.

Node Index

Common Lua Globals and Functions

- File location: `Game/Scripts/common.lua`
- Loaded from: `Game/Scripts/main.lua`

Globals

Use the following globals to avoid temporary Lua memory allocations:

Name	Description
g_SignalData_point	Basic 3D vector value used by **g_SignalData.**
g_SignalData_point2	Basic 3D vector value used by **g_SignalData.**
g_SignalData	Used to pass signal data in AI behavior scripts (see: Signals (p. 86)).
g_StringTemp1	Commonly used for temporary strings inside Lua functions.

Name	Description
g_HitTable	Commonly used by the **Physics.RayWorldIntersection** function.

A **g_HitTable** used with **Physics.RayWorldIntersection** can contain the following parameters:

Parameter	Description
pos	3D vector world coordinates of the ray hit.
normal	3D normal vector of the ray hit.
dist	Distance of the ray hit.
surface	Type of surface hit.
entity	Script table of entity hit (if one was hit).
renderNode	Script handle to a foliage or static render node.

A **g_SignalData** table can contain the following parameter types:

Type	Description
Vec3	3D vector.
ScriptHandle	Normally used to pass along an entity ID.
Floating Point	Floating point value.
Integer	Integer or number value.
String	String value.

AIReload()

Reloads the **aiconfig.lua** Lua script (`Game/Scripts/AI/`).

AIDebugToggle()

Toggles the ai_DebugDraw console variable on and off.

ShowTime()

Logs the current system time to the console. Format is Day/Month/Year, Hours:Minutes.

count()

Returns the number of key-value pairs in a given table.

Parameter	Description
_tbl	Table to retrieve the number of key-value pairs from.

new()

Creates a new table by copying an specified existing table. This function is commonly used to create a local table based on an entity parameter table.

Parameter	Description
_obj	Existing table you want to create a new one from.
norecurse	Flag indicating whether or not to recursively recreate all sub-tables. If set to TRUE, sub-tables will not be recreated.

merge()

Merges two tables without merging functions from the source table.

Parameter	Description
dst	Destination table to merge source table information into.
src	Source table to merge table information from.
recurse	Flag indicating whether or not to recursively merge all sub-tables.

mergef()

Merges two tables including merging functions from the source table.

Parameter	Description
dst	Destination table to merge source table information into.
src	Source table to merge table information from.
recursive	Flag indicating whether or not to recursively merge all sub-tables.

Vec2Str()

Converts a 3D vector table into a string and returns it in the following format: (x: X.XXX y: Y.YYY z: Z.ZZZ).

Parameter	Description
vec	3D vector table to convert. Example: `{x=1,y=1,z=1}`.

LogError()

Logs an error message to the console and the log file. Message appears in red text in the console.

Parameter	Description
fmt	Formatted message string.

Parameter	Description
...	Optional argument list. For example: `LogError("MyError: %f", math.pi);`

LogWarning()

Logs a warning message to the console and the log file. Message appears in yellow text in the console.

Parameter	Description
fmt	Formatted message string.
...	Optional argument list. For example: `LogWarning("MyError: %f", math.pi);`

Log()

Logs a message to the console and the log file. Commonly used for debugging purposes.

Parameter	Description
fmt	Formatted message string.
...	Optional argument list. For example: `Log("MyLog: %f", math.pi);`

dump()

Dumps information from a specified table to the console.

Parameter	Description
_class	Table to dump to console. For example: `g_localActor`
no_func	Flag indicating whether or not to dump the table functions.
depth	Depth of the tables tree dump information from.

EmptyString()

Checks whether or not a given string is set and its length is greater than zero. Returns `TRUE` or `FALSE`.

Parameter	Description
str	String to check for.

NumberToBool()

Checks whether or not a number value is true (non-zero) or false (zero).

Parameter	Description
n	Number to check for.

EntityName()

Retrieves the name of a specified entity ID or entity table. If the entity doesn't exist, this function returns an empty string.

Parameter	Description
entity	Entity table or entity ID to return a name for.

EntityNamed()

Checks whether or not an entity with the specified name exists in the entity system. Returns `TRUE` or `FALSE`. Commonly used for debugging.

Parameter	Description
name	Name of entity to check for.

SafeTableGet()

Checks whether or not a sub-table with a specified name exists in a table. If the sub-table exists, this function returns it; otherwise the function returns `nil`.

Parameter	Description
table	Table to check for the existence of a sub-table.
name	Sub-table name to check for.

EntityUtils Lua Functions

This topic describes the commonly used Lua entity utility functions.

- File location: `Game/Scripts/Utils/EntityUtils.lua`
- Loaded from: `Game/Scripts/common.lua`

DumpEntities()

Dumps to console all entity IDs, names, classes, positions, and angles that are currently used in a map. For example:

```
[userdata: 00000002]..name=Grunt1 clsid=Grunt pos=1016.755,1042.764,100.000
ang=0.000,0.000,1.500
[userdata: 00000003]..name=Grunt2 clsid=Grunt pos=1020.755,1072.784,100.000
ang=0.000,0.000,0.500
...
```

CompareEntitiesByName()

Compares two entities identified by name. This function is commonly used when sorting tables.

Parameter	Description
ent1	Name of first entity table.
ent2	Name of second entity table.

Example

```
local entities = System.GetEntitiesByClass("SomeEntityClass");
table.sort(entities, CompareEntitiesByName);
```

CompareEntitiesByDistanceFromPoint()

Compares the distance of two entities from a specified point. If the distance is greater for Entity 1 than for Entity 2 (that is, Entity 1 is further away), this function returns `TRUE`, otherwise it returns `FALSE`.

Parameter	Description
ent1	Entity 1 table
ent2	Entity 2 table
point	3D position vector identifying the point to measure distance to.

Example

```
local ent1 = System.GetEntityByName("NameEntityOne");
local ent2 = System.GetEntityByName("NameEntityTwo");

if(CompareEntitiesByDistanceFromPoint( ent1, ent2, g_localActor:GetPos()))then

   Log("Entity One is further away from the Player than Entity two...");
end
```

BroadcastEvent()

Processes an entity event broadcast.

Parameter	Description
sender	Entity that sent the event.

Parameter	Description
event	String based entity event to process.

Example

```
BroadcastEvent(self, "Used");
```

MakeDerivedEntity()

Creates a new table that is a derived version of a parent entity table. This function is commonly used to simplify the creation of a new entity script based on another entity.

Parameter	Description
_DerivedClass	Derived class table.
_Parent	Parent or base class table.

MakeDerivedEntityOverride()

Creates a new table that is a derived class of a parent entity. The derived table's properties will override those from the parent.

Parameter	Description
_DerivedClass	Derived class table.
_Parent	Parent or base class table.

MakeUsable()

Adds usable functionality, such as an **OnUsed** event, to a specified entity.

Parameter	Description
entity	Entity table to make usable.

Example

```
MyEntity = { ... whatever you usually put here ... }

MakeUsable(MyEntity)

function MyEntity:OnSpawn() ...

function MyEntity:OnReset()
  self:ResetOnUsed()
  ...
end
```

MakePickable()

Adds basic "pickable" functionality to a specified entity. The `bPickable` property is added to the entity's properties table.

Parameter	Description
entity	Entity table to make pickable.

MakeSpawnable()

Adds spawn functionality to a specified entity. Commonly used for **AI** actors during creation.

Parameter	Description
entity	Entity table to make spawnable.

EntityCommon.PhysicalizeRigid()

Physicalizes an entity based on the specified entity slot and its physics properties.

Parameter	Description
entity	Entity table to physicalize.
nSlot	Entity slot to physicalize.
Properties	Physics properties table
bActive	Not used.

Math Lua Globals and Functions

This topic describes the commonly used math global vectors, constants, and functions.

- File location: `Game/Scripts/Utils/Math.lua`
- Loaded from: `Game/Scripts/common.lua`

Global Vectors

The following globals should be used to avoid temporary Lua memory allocations:

Global Name	Description
g_Vectors.v000	Basic zero vector.
g_Vectors.v001	Positive z-axis direction vector.
g_Vectors.v010	Positive y-axis direction vector.
g_Vectors.v100	Positive x-axis direction vector.

Global Name	Description
g_Vectors.v101	The x and z-axis direction vector.
g_Vectors.v110	The x and y-axis direction vector.
g_Vectors.v111	The x, y and z-axis vector.
g_Vectors.up	Positive z-axis direction vector.
g_Vectors.down	Negative z-axis direction vector.
g_Vectors.temp	Temporary zero vector.
g_Vectors.tempColor	Temporary zero vector. Commonly used for passing rgb color values.
g_Vectors.temp_v1	Temporary zero vector.
g_Vectors.temp_v2	Temporary zero vector.
g_Vectors.temp_v3	Temporary zero vector.
g_Vectors.temp_v4	Temporary zero vector.
g_Vectors.vec-MathTemp1	Temporary zero vector.
g_Vectors.vec-MathTemp2	Temporary zero vector.

Constants

Constant Name	Description
g_Rad2Deg	Basic radian-to-degree conversion value.
g_Deg2Rad	Basic degree-to-radian conversion value.
g_Pi	Basic Pi constant based on `math.pi`.
g_2Pi	Basic double-Pi constant based on `math.pi`.
g_Pi2	Basic half-Pi constant based on `math.pi`.

IsNullVector()

Checks whether or not all components of a specified vector are null.

Parameter	Description
a	Vector to check.

IsNotNullVector()

Checks whether or not any components of a specified vector is not null.

Parameter	Description
a	Vector to check.

LengthSqVector()

Retrieves the squared length of a specified vector.

Parameter	Description
a	Vector to retrieve length for.

LengthVector()

Retrieves the length of a specified vector.

Parameter	Description
a	Vector to retrieve length for.

DistanceSqVectors()

Retrieves the squared distance between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

DistanceSqVectors2d()

Retrieves the squared distance between two vectors in 2D space (without z-component).

Parameter	Description
a	First vector.
b	Second vector.

DistanceVectors()

Retrieves the distance between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

dotproduct3d()

Retrieves the dot product between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

dotproduct2d()

Retrieves the dot product between two vectors in 2D space (without z-component).

Parameter	Description
a	First vector.
b	Second vector.

LogVec()

Logs a specified vector to console.

Parameter	Description
name	Descriptive name of the vector.
v	Vector to log.

Example

```
LogVec("Local Actor Position", g_localActor:GetPos())
```

Console output:

```
<Lua> Local Actor Position = (1104.018066 1983.247925 112.769440)
```

ZeroVector()

Sets all components of a specified vector to zero.

Parameter	Description
dest	Vector to zero out.

CopyVector()

Copies the components of one vector to another.

Parameter	Description
dest	Destination vector.
src	Source vector.

SumVectors()

Adds up the components of two vectors.

Parameter	Description
a	First vector.
b	Second vector.

NegVector()

Negates all components of a specified vector.

Parameter	Description
a	Vector to negate.

SubVectors()

Copies the componentwise subtraction of two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Second vector.

FastSumVectors()

Copies the componentwise addition of two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Second vector.

DifferenceVectors()

Retrieves the difference between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

FastDifferenceVectors()

Copies the componentwise difference between two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Second vector.

ProductVectors()

Retrieves the product of two vectors.

Parameter	Description
a	First vector.
b	Second vector.

FastProductVectors()

Copies the product of two vectors to a destination vector.

Parameter	Description
dest	Destination vector.

Parameter	Description
a	First vector.
b	Second vector.

ScaleVector()

Scales a specified vector *a* by a factor of *b*.

Parameter	Description
a	Vector.
b	Scalar.

ScaleVectorInPlace(a,b)

Retrieves a new vector based on a copy of vector *a* scaled by a factor *b*.

Parameter	Description
a	First vector.
b	Scalar.

ScaleVectorInPlace(dest,a,b)

Copies vector *a* scaled by the factor of *b* to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Scalar.

NormalizeVector()

Normalizes a specified vector.

Parameter	Description
a	Vector to normalize.

VecRotate90_Z()

Rotates a specified vector by 90 degree around the z-axis.

Parameter	Description
v	Vector to rotate.

VecRotateMinus90_Z()

Rotates a specified vector by -90 degree around the z-axis.

Parameter	Description
v	Vector to rotate.

crossproduct3d()

Copies the result of the cross product between two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
p	First vector.
q	Second vector.

RotateVectorAroundR()

Copies to a destination vector the result of the vector rotation of vector *p* around vector *r* by a specified angle.

Parameter	Description
dest	Destination vector.
p	First vector.
r	Second vector.
angle	Rotation angle.

ProjectVector()

Copies to a destination vector the result of the vector projection of vector *P* to the surface with a specified normal *N*.

Parameter	Description
dest	Destination vector.

Parameter	Description
P	Vector to project.
N	Surface normal.

DistanceLineAndPoint()

Retrieves the distance between point *a* and the line between *p* and *q*.

Parameter	Description
a	Point to measure from.
p	Vector p.
q	Vector q.

LerpColors()

Performs linear interpolation between two color/vectors with a factor of *k*.

Parameter	Description
a	Color/vector a.
b	Color/vector b.
k	Factor k.

Lerp()

Performs linear interpolation between two scalars with a factor of *k*.

Parameter	Description
a	Scalar a.
b	Scalar b.
k	Factor k.

__max()

Retrieves the maximum of two scalars.

Parameter	Description
a	Scalar a.
b	Scalar b.

__min()

Retrieves the minimum of two scalars.

Parameter	Description
a	Scalar a.
b	Scalar b.

clamp()

Clamps a specified number between minimum and maximum.

Parameter	Description
_n	Number to clamp.
_min	Lower limit.
_max	Upper limit.

Interpolate()

Interpolates a number to a specified goal by a specified speed.

Parameter	Description
actual	Number to interpolate.
goal	Goal.
speed	Interpolation speed.

sgn()

Retrieves the sign of a specified number (0 returns 0).

Parameter	Description
a	Number to get sign for.

sgnnz()

Retrieves the sign of a specified number (0 returns 1).

Parameter	Description
a	Number to get sign for.

sqr()

Retrieves the square of a specified number.

Parameter	Description
a	Number to square.

randomF()

Retrieves a random float value between two specified numbers.

Parameter	Description
a	First number.
b	Second number.

iff()

Checks the condition of a test value and returns one of two other values depending on whether the test value is `nil` or not.

Parameter	Description
c	Test value.
a	Return value if test value is not nil.
b	Return value if test value is nil.

Physics Lua Functions

These functions are commonly used to register new explosion and crack shapes in the physics engine.

File location: `Game/Scripts/physics.lua`

- Loaded from: `Game/Scripts/main.lua`

Physics.RegisterExplosionShape()

Registers a boolean carving shape for breakable objects in the physics engine.

Parameter	Description
sGeometryFile	Name of a boolean shape cgf file.
fSize	Shape's characteristic size.
BreakId	Breakability index (0-based) used to identify the breakable material.

Parameter	Description
fProbability	Shape's relative probability; when several shapes with the same size appear as candidates for carving, these relative probabilities are used to select one.
sSplintersfile	Name of a splinters cgf file, used for trees to add splinters at the breakage location.
fSplintersOffset	Size offset for the splinters.
sSplintersCloudEffect	Name of splinters particle fx; this effect is played when a splinters-based constraint breaks and splinters disappear.

Physics.RegisterExplosionCrack()

Registers a new explosion crack for breakable objects in the physics engine.

Parameter	Description
sGeometryFile	Name of a crack shape cgf file. This type of file must have three helpers to mark the corners, named "1","2" and "3".
BreakId	Breakability index (0-based) used to identify the breakable material.

Integrating Lua and C++

The CryScript system abstracts a Lua virtual machine for use by the other systems and the game code. It includes the following functionality:

- calling script functions
- exposing C++-based variables and functions to scripts
- creating script tables stored in virtual machine memory

The CryScript system is based on Lua 5. More information on the Lua language can be found at http://www.lua.org.

Accessing Script Tables

A global script table can be retrieved by calling `IScriptSystem::GetGlobalValue()`. The IScriptTable is used to represent all script tables/variables.

Exposing C++ Functions and Values

To expose C++ functions and variables to scripts, you'll need to implement a new class. The easiest way is to derive the `CScriptableBase` class, which provides most of the functionality.

Exposing Constants

To expose constant values to scripts, use the `IScriptSystem::SetGlobalValue()`. For example, to expose a constant named MTL_LAYER_FROZEN to our scripts, use the following code:

```
gEnv->pScriptSystem->SetGlobalValue("MTL_LAYER_FROZEN", MTL_LAYER_FROZEN);
```

Exposing Functions

To expose C++ functions to scripts, implement a new class derives from `CScriptableBase`, as shown in the following example.

```
classCScriptBind_Game :
    publicCScriptableBase
{
public:
    CScriptBind_Game( ISystem* pSystem );
    virtual ~CScriptBind_Game() {}

    intGameLog(IFunctionHandler* pH, char* pText);
};
```

Add the following code inside the class constructor:

```
Init(pSystem->GetIScriptSystem(), pSystem);
SetGlobalName("Game");

#undef SCRIPT_REG_CLASSNAME
#define SCRIPT_REG_CLASSNAME &CScriptBind_Game::

SCRIPT_REG_TEMPLFUNC(GameLog, "text");
```

In a Lua script, you can access your new ScriptBind function as follows:

```
Game.GameLog("a message");
```

Lua Script Usage

Lumberyard uses Lua for its scripting language.

The Entity system can attach a script proxy to any entity, which is in the form of a table that can include data and functions. AI behaviors are often written in scripts. Additionally, several game systems, including Actor, Item, Vehicle, and GameRules, rely on scripting to extend their functionality.

The advantages of using scripts include:

- Fast iteration – Scripts can be reloaded within the engine.
- Runtime performance – Careful usage of available resources can result into scripts that run nearly as fast as compiled code.
- Easy troubleshooting – An embedded Lua debugger can be invoked at any time.

Most of the systems in Lumberyard expose ScriptBind functions, which allow Lua scripts to call existing code written in C++.

Running Scripts

You can run scripts either by calling script files directly from code or by using console commands.

In code

Scripts are stored in the `\Game\Scripts` directory. To invoke a script file, call the `LoadScript` function from your C++ code. For more information, see Integrating Lua and C++ (p. 330). Another option is to create a script entity, as described in Entity Scripting (p. 271).

In the Console

Script instructions can be executed using the in-game console. This can be done by appending the **#** character before the instructions. This functionality is limited to Lumberyard Editor or when running the launcher in dev mode (using the `-DEVMODE` command-line argument).

Reloading Scripts During Runtime

In Lumberyard Editor it is always possible to reload entities within the user interface. When reloading a script entity, choose the **Reload Script** button, which is found in the Rollup Bar.

You can also use the following ScriptBind functions to reload scripts.

- `Script.ReloadScript(filename)`
- `Script.ReloadScripts()`

To invoke these functions from the console, use the following syntax:

```
#Script.ReloadScript("Scripts\\EntityCommon.lua")
```

Using the Lua Remote Debugger

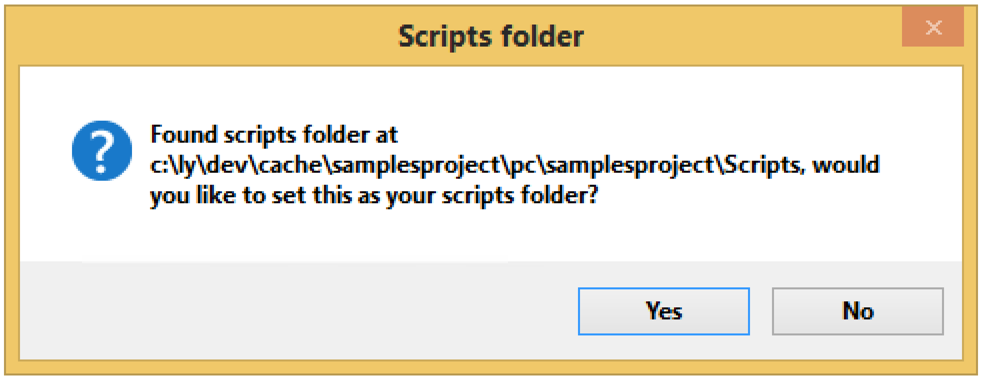
Lumberyard includes a standalone visual script debugger for Lua. To start the debugger, you first enable it in the console, and then run the `LuaRemoteDebugger.exe` executable file.

1. In the Lumberyard Editor console or game console, type **`lua_debugger 1`** or **`lua_debugger 2`**. This enables enable debugging in one of the following two modes:
   - Mode 1 – The debugger breaks on both breakpoints and script errors.
   - Mode 2 – The debugger breaks only on script errors.
2. Run the Lua remote debugger executable file at the Lumberyard folder location `\dev\Tools\LuaRemoteDebugger\LuaRemoteDebugger.exe`.
3. In the Lua remote debugger, on the **File** menu, choose **Connect**.
4. If you are running the game in the editor (you pressed **`Ctrl-G`**) and want to debug your scripts, choose **PC (Editor)**. If you want to attach the debugger to the built game executable, choose **PC (Game)**.

   For **IP address** and **Port**, type the IP address and port of the computer to which you want to connect. The default options connect to the game on your local computer. The default IP address is 127.0.01 (localhost). For **PC (Editor)**, the default port is 9433. For **PC (Game)**, the default port is 9432.

5. Choose **Connect**. In Lumberyard Editor, the console window displays **Lua remote debug client connected**.

   The first time you run Lua remote debugger, it prompts you for the scripts folder:

   Found scripts folder at c:\ly\dev\cache\samplesproject\pc\samplesproject\Scripts, would you like to set this as your scripts folder?

   The default folder is the `Scripts` folder of the project that you are running. For example, if you are running the samples project, the folder is `samplesproject/Scripts`.
6. To accept the default location, click **Yes**.

   Note
   To change the scripts folder location, choose **File**, **Set Scripts Folder**.

   After you choose the location for your scripts folder, the folder's contents are shown in the navigation tree on the left.

Performing Tasks in the Lua Remote Debugger

To perform specific tasks in the Lua remote debugger, see the following table:

To do this	Do this
Open a script file	Double click the script file in the navigation tree, or press **`Ctrl+O`** to open the **Find File** dialog.
Set a break point	Place the cursor on the line in the script where you want the break to occur, and then click the red dot in the toolbar or press **`F9`**. When program execution stops on a break point, the **Call Stack** and **Locals** tabs populate.
Remove a break point	Place the cursor on the line with the breakpoint that you want to remove, and then click the red dot in the toolbar or press **`F9`**.
Use the Breakpoints tab	The **Breakpoints** tab window displays each of your breakpoints with a checkbox next to it. To enable or disable a breakpoint, select or clear its checkbox. In the script window, the breakpoint's status is indicated by its color: red is active; gray is disabled.
To watch (inspect) variable values	When execution is paused on a breakpoint, click the **Watch** tab, click the first column of a blank row, and then type the name of the variable that you want to watch.
Pause execution	Click the pause (break) button on the toolbar or press **`Ctrl+Alt+Pause`**.
Resume execution	Click the play button on the toolbar or press **`F5`**.

To do this	Do this
Step over a procedure	Click the toolbar icon or press **`F10`**.
Step into a procedure	Click the toolbar icon or press **`F11`**.
Step out of a procedure	Click the toolbar icon or press **`Shift+F11`**.
Close a script file	Choose **File**, **Close**, or press **`Ctrl+W`**
Disconnect from the editor or game	In the Lua debugger, choose **File**, **Disconnect**. The Lumberyard console displays a **network connection terminated** message.

Note
Code changes that you make in the debugger window do not change the loaded script and are discarded after the debugger window is closed.

Using the Lua XML Loader

There is a generic interface for parsing and translating XML files into Lua files. This interface uses an XML file as a definition format that declares what kind of XML is included in a file and what kind of Lua to create from the XML. The format includes some simple validation methods to ensure that the data received is what is expected.

XML Data

The XML loader can distinguish between three kinds of data: properties, arrays, and tables.

Tables

This table represents a Lua-based table:

```
letters = { a="a", b="b", c="c" };
```

In an XML data file, this table would look like this:

```
<letters a="a" b="b" c="c"/>
```

The XML definition file would look like this:

```
<Table name="letters">
  <Property name="a" type="string"/>
  <Property name="b" type="string"/>
  <Property name="c" type="string"/>
</Table>
```

Each element can be marked as optional in the definition file using the attribute `optional="1"`.

Arrays

There are two possible types of arrays. The first type is a simple group of elements, shown in Lua like this:

```
numbers = {0,1,2,3,4,5,6,7,8,9}
```

In the XML data file, the array would look like this:

```
<numbers>
  <number value="0"/>
  <number value="1"/>
  <number value="2"/>
  <number value="3"/>
  <number value="4"/>
  <number value="5"/>
  <number value="6"/>
  <number value="7"/>
  <number value="8"/>
  <number value="9"/>
</numbers>
```

The data definition file would look like this:

```
<Array name="numbers" type="int" elementName="number"/>
```

The second array type is an array of tables. In Lua:

```
wheels = {
  {size=3, weight=10},
  {size=2, weight=1},
  {size=4, weight=20},
}
```

In the XML data file:

```
<wheels>
  <wheel size="3" weight="10"/>
  <wheel size="2" weight="1"/>
  <wheel size="4" weight="20"/>
</wheels>
```

The XML definition file:

```
<Array name="wheels" elementName="wheel"> <!-- note no type is attached -->
  <Property name="size" type="float"/>
  <Property name="weight" type="int"/>
</Array>
```

Loading and Saving a Table from Lua

To load and initialize a Lua table:

```
someTable = CryAction.LoadXML( definitionFileName, dataFileName );
```

When storing XML files for scripts, the recommended practice is to keep the definition files with the scripts that use them, but store the data files in a directory outside the Scripts directory.

To save a table from Lua:

```
CryAction.SaveXML( definitionFileName, dataFileName, table );
```

Data Types

The following data types are available, and can be set wherever a "type" attribute is present in the definition file.

- float – Floating point number.
- int – Integer.
- string – String.
- bool – Boolean value.
- Vec3 – Floating point vectors with three components. Values of this type are expressed as follows:
  - XML – **`"1,2,3"`**
  - Lua – **`{x=1,y=2,z=3}`**

Enums

For string type properties, an optional `<Enum>` definition can be used. Property values will be validated against the enum.

Example:

```
<Property name="view" type="string">
 <Enum>
  <Value>GhostView</Value>
  <Value>ThirdPerson</Value>
  <Value>BlackScreen</Value>
 </Enum>
</Property>
```

Enum support for other data types can be added, if necessary.

Example

XML definition file:

```
<Definition root="Data">

 <Property name="version" type="string"/>
 <Table name="test">
  <Property name="a" type="string"/>
  <Property name="b" type="int" optional="1"/>
  <Array name="c" type="string" elementName="Val"/>
  <Array name="d" elementName="Value">
     <Property name="da" type="float"/>
     <Property name="db" type="Vec3"/>
  </Array>
  <Property name="e" type="int"/>
 </Table>

</Definition>
```

Corresponding XML data file:

```
<Data version="Blag 1.0">
 <test
  a="blag"
  e="3">
  <c>
     <Val value="blag"/>
     <Val value="foo"/>
  </c>
  <d>
     <Value da="3.0" db="2.1,2.2,2.3"/>
     <Value da="3.1" db="2.1,2.2,2.3"/>
     <Value da="3.2" db="3.1,3.2,2.3"/>
  </d>
 </test>
</Data>
```

Recommended Reading

The following resources on the Lua language are recommended reading when working with scripts with Lumberyard.

- Lua 5.1 Reference Manual
- Programming in Lua, Third Edition
- Other books

Networking System

GridMate is Lumberyard's networking subsystem. GridMate is designed for efficient bandwidth usage and low-latency communications. You can synchronize objects over the network with GridMate's replica framework. GridMate's session management integrates with major online console services and lets you handle peer-to-peer and client-server topologies with host migration. GridMate also supports in-game achievements, leaderboards, and cloud-based saved games through third-party social services such as Xbox Live, PlayStation Network, and Steam. For an example of how to set up a multiplayer project, see Multiplayer Sample Project in the Amazon Lumberyard User Guide.

This section discusses the various components of, and setup requirements for, your Amazon Lumberyard game networking environment.

Topics

Setting up a Multiplayer Server

After you create and set up a multiplayer level and export the level to Lumberyard, you can run a server to host your new level.

Before you can host your new level, you must do the following:

- Ensure the same SDK build is installed on all the computers you will use to play the level.
- Copy the multiplayer level folder to each SDK installation where you will play.

  Example: `\root\Game\Levels`
- Identify the IP address of your host server, which will be used to allow other players to join your game.

Hosting a Level

Hosting a level on your server takes just a few steps.

To host a level on the server

1. On your host server, run the game executable.
2. From the console, run the command(s) for the task(s) you want to perform.
   - `mpstart` *`local_port`* – Initializes the multiplayer service. Replace *`local_port`* with the local UDP port you want to use to initialize the socket (optional). Port 64090 is used by default.
   - `mphost` – Hosts a session. The server listens for incoming connections on the port that you specified in `mpstart`.
   - `map` *`map_name`* – Launches the specified map. Replace *`map_name`* with the name of the map you want to use.

Connecting to the Server

After the server is running, you can connect to the hosted game from each of the computers you want to use to play the multiplayer game.

To connect to the server

1. On your computer, run the game executable.
2. From the console, run the commands for the tasks that you want to perform.
   - `mpstart` *`local_port`* – Initializes the multiplayer service. Replace *`local_port`* with the local UDP port you want to use to initialize the socket (optional). Port 64090 is used by default. To connect to a server on the same computer, replace local_port with 0 to use an ephemeral port.

     Note
     When running a client on the same computer as the server, set the client to use an ephemeral port. Call `mpstart 0` in order to prevent port conflicts.
   - `mpjoin` *`server_addr`* *`server_port`* – Join a game instance. The *`server_addr`* and *`server_port`* settings are optional. Localhost:64090 is used by default.

Console Command Summary

Use the following commands when working with a network server.

`gm_debugdraw debug_draw_level`

Set the debug draw level. Accepts as a parameter a number whos bits represent the flags for the debug data to draw. For example, when set to 1, displays an overlay with GridMate network statistics and information.

The available bit flags come from the enum `DebugDrawBits` and are as follows:

```
enum DebugDrawBits
        {
            Basic           = BIT(0),
            Trace           = BIT(1),
            Stats           = BIT(2),
            Replicas        = BIT(3),
            Actors          = BIT(4),
            EntityDetail    = BIT(5),

            Full            = Basic | Trace | Stats | Replicas | Actors,
            All             = 0xffffffff,
        };
```

gm_disconnectDetection
: When set to 0, disables disconnect detection. This is useful when you are debugging a server or client and don't want to be disconnected when stepping through code. The default value is 1.

gm_dumpstats
: Write GridMate network profiling stats to file.

gm_dumpstats_file
: The file to which GridMate profiling stats are written. The default is `net_profile.log`.

gm_net_simulator
: Activate GridMate network simulator to simulate latency, packet loss, bandwidth restrictions, and other conditions. For available options, type `gm_net_simulator help` .

gm_setdebugdraw
: Display an overlay with detailed GridMate networking statistics and information. A user-friendly helper command for **`gm_debugdraw debug_draw_level`**. Possible parameters are `Basic`, `Trace`, `Stats`, `Replicas`, and `Actors`.

gm_stats_interval_msec
: Set the interval, in milliseconds, for gathering network profiling statistics. The default is 1000.

gm_tracelevel trace_level
: Set the GridMate debugging trace verbosity level. The default is 0. The higher the value, the greater the verbosity. Typical values range from 1 to 3.

mpstart [*local_port*]
: Initialize the network system and optionally set the local UDP port that initializes the socket. The default port is 64090. To use the ephemeral port, set the port to 0. This is useful if you want to connect to a server on the same computer as the client.

mphost
: Create a session as host. The server listens for incoming connections on the port specified in `mpstart`.

mpjoin [*server_addr*] [*server_port*]
: Connect to a server at the optionally specified *`server_addr`* and *`server_port`*. The defaults are `localhost` and `64090`, respectively.

map *map_name*
: Loads the level with the specified map name. Replace *`map_name`* with the name of the map you want to use. To view a list of available levels, type **`map`**, and then press the tab key.

mpdisconnect
: Terminate the current game instance session.

mpstop
: Terminate the multiplayer service.

Sample Project

For an example of how to set up a multiplayer project, see Multiplayer Sample Project in the Amazon Lumberyard User Guide

Network Serialization and Aspects

All objects that are intended to be synchronized over the network should have a function called `NetSerialize()`. In the `GameObject`, this appears as: `IGameObject::NetSerialize()`.

The `NetSerialize()` function uses a `TSerialize` object of type `ISerialize` to transform relevant data to a stream. The serialization uses different aspects and profiles to distinguish the various types of streams.

Note
Serialized data for a given aspect and profile must remain fixed. For example, if you serialized four floats, you must always serialize four floats.

Aspects

You use aspects to logically group data together.

Aspects are defined as follows:

- `eEA_GameClient` – Information sent from the client to the server, if the client has authority over the object.
- `eEA_GameServer` – The normal server to client data stream.
- `Dynamic/Static` – Data that is constantly changing should be added to the `Dynamic` aspect. Objects that rarely change should be added to the `Static` aspect. Updates are not sent if only one value changes.
- `eEA_Script` – Used where script network data is transported, including any script RMI calls.
- `eEA_Physics` – Used where physics data is transported. It is not divided into client/server because it always uses the same path: (controlling-client) to serve other clients.

Profiles

Profiles allow an aspect's fixed format data to be different. There are potentially eight profiles per aspect, and they are only used for physics aspects (for example, switching between ragdoll and living entity).

RMI Functions

To send remote method invocations (RMIs), use the `InvokeRMI` function, which has the following syntax:

```
void InvokeRMI( IRMIRep& rep, ParamsType&& params, uint32 where, ChannelId
channel = kInvalidChannelId );
```

Parameters

`rep`
Represents the remote function to be called (the RMI ID).

`params`
Specifies the parameters to pass into the remote function.

`where`
Specifies a flag that determines the category of clients to which the RMI will be sent. For information, see the RMI Function Flags (p. 342) section later in this document.

`channel`
: Specifies specific clients to which the RMI will be sent, or specific clients to exclude. For information, see the RMI Function Flags (p. 342) section later in this document.

Ordering RMI Functions

The `IGameObject.h` file includes macros for declaring RMI classes (for example, those beginning with `DECLARE_SERVER_RMI_...`). The different declaration types are as follows:

- `PREATTACH` – The RMI is attached at the top of the data update for the object. You can use this declaration type to prepare the remote entity for new incoming data.
- `POSTATTACH` – The RMI is attached at the bottom of the data update, so it is called after the data is serialized. You can use this declaration type to complete an action with the new data.
- `NOATTACH` – The RMI is not attached to a data update, so the RMI cannot rely on the data. You can use this declaration type for calls that don't rely on data.

Ordering Rules

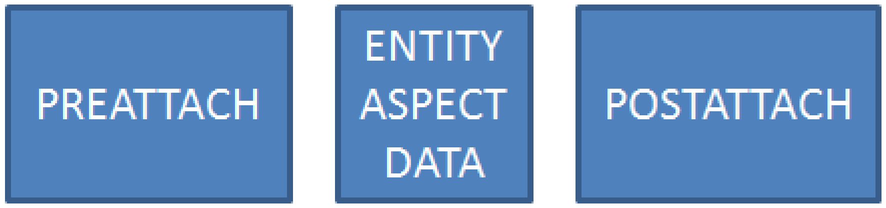
The order for RMIs is only applicable within an object and attachment type set.

For example, in the following ordered list, PLAYER RMI 1, 2, and 3 will arrive in that order; however, ITEM RMI 1 might arrive before or after the following PLAYER RMIs:

- PLAYER RMI 1
- PLAYER RMI 2
- ITEM RMI 1
- PLAYER RMI 3

Using declaration types adds a layer of complication to the order of incoming data:

- `PREATTACH` – Messages are ordered within themselves.
- `POSTATTACH` – Messages are ordered within themselves.
- `NOATTACH` – Messages are ordered within themselves; however, `NOATTACH` can only fall on either side of the following diagram and never in between:



RMI Function Flags

To specify the clients that will receive an RMI, replace the *`where`* parameter in the `InvokeRMI` function with one of the following flags.

Server RMIs

`eRMI_ToClientChannel`
Sends an RMI from the server to a specific client. Specify the destination channel in the *`channel`* parameter.

`eRMI_ToOwningClient`
Sends an RMI from the server to the client that owns the actor.

`eRMI_ToOtherClients`
Sends an RMI from the server to all clients except the client specified. Specify the client to ignore in the *`channel`* parameter.

`eRMI_ToRemoteClients`
Sends an RMI from the server to all remote clients. Ignores the local client.

`eRMI_ToOtherRemoteClients`
Sends an RMI from the server to all remote clients except the remote client specified. Ignores the local client. The remote client to ignore is specified in the *`channel`* parameter.

`eRMI_ToAllClients`
Sends an RMI from the server to all clients.

Client RMIs

`eRMI_ToServer`
Sends an RMI from the client to the server.

Examples

To define a function to be implemented as RMI, use the `IMPLEMENT_RMI #define` from `IGameObject.h`.

```
#define IMPLEMENT_RMI(cls, name)
```

The following example implements a new function called `Cl_SetAmmoCount` in the `CInventory` class to be used as a client-side RMI, taking one argument of type `TRMIInventory_Ammo`:

```
Class CInventory : public CGameObjectExtensionHelper<CIventory, IInventory>
{
  // …
  DECLARE_CLIENT_RMI_NOATTACH(Cl_SetAmmoCount, TRMIInventory_Ammo, eNRT_Reli
ableOrdered);
  // …
};

IMPLEMENT_RMI(CInventory, Cl_SetAmmoCount)
{
  // Game code:
  TRMIInventory_Ammo Info(params);
  IEntityClass* pClass = gEnv->pEntitySystem->GetClassRegistry()->Find
Class(Info.m_AmmoClass.c_str());
  If (pClass)
    SetAmmoCount(pClass, Info.m_iAmount);

  return true;    // Always return true - false will drop connection
}
The following line will invoke the function:
```

```
pInventory->GetGameObject()->InvokeRMI(CInventory::Cl_SetAmmoCount(), TRMIIn
ventory_Ammo(“Pistol”, 10), eRMI_ToAllClients);
```

The following line will invoke the function:

```
pInventory->GetGameObject()->InvokeRMI(CInventory::C1_SetAmmoCount(), TRMIIn
ventory_Ammo("Pistol", 10), eRMI_ToAllClients);
```

Session Service

GridMate's session service provides session connectivity and management. Both hub-and-spoke (client/server) and P2P full-mesh topologies are supported, and multiple sessions can be created for each GridMate instance.

Each session creates its own carrier and replica manager instances, so there is no interaction between sessions. GridMate sessions support host migration when running in P2P mode.

Starting and Stopping the Session Service

Start the session service in GridMate by calling `IGridMate::StartMultiplayerService()` or `IGridMate::StartMultiplayerServiceCustom()`. Note that `GridMate::OnlineService` must be started before the session service can be started.

To stop the service, call `IGridMate::StopMultiplayerService()`. This terminates all current sessions and searches.

Start session service example

The following LAN multiplayer service example starts the online service if it has not already been started.

```
IGridMate* gridmate = gEnv->pNetwork->GetGridMate();
If (gridmate)
{
    // start the online service if needed
    if (!gridmate->IsOnlineServiceStarted())
    {
       gridmate->StartOnlineService(GridMate::ST_LAN, GridMate::OnlineService
Desc());
    }

   gridmate->StartMultiplayerService(GridMate::ST_LAN, GridMate::SessionServi
ceDesc(), nullptr);
}
```

Hosting a Session

To host a session, call `SessionService::HostSession()`. The function takes two arguments: `SessionParams` and `CarrierDesc`. The function returns a `GridSession` pointer.

`SessionParams` is used to configure the sessions topology, the number of slots, and the matchmaking properties. The local user to which to bind the current session is also set using `SessionParams`.

`CarrierDesc` is used to set the carrier parameters used for the current session.

The pointer returned by `SessionService::HostSession()` will remain valid until the `GridSessionCallbacks::OnSessionDelete()` event occurs.

Host session example

The following is an example component that creates a GridMate session and listens for clients that connect to the session.

```
// GridMateHost
#include <AZCore/Component/Component.h>
#include <AZCore/Component/TickBus.h>
#include <GridMate/GridMate.h>
#include <GridMate/Session/Session.h>
#include <GridMate/Session/LANSession.h>

class HostComponent
    : public AZ::Component
    , public AZ::TickBus::Handler
    , public GridMate::SessionEventBus::Handler
{
    GridMate::IGridMate* mGridMate;
    GridMate::GridSession* mSession;
public:
    AZ_COMPONENT(HostComponent, "{41900198-1122-4392-A579-CDDC85A23277}");

    HostComponent() : mSession(nullptr), mGridMate(nullptr) {}

    // Events from GridMate::SessionEventBus
    void OnSessionCreated(GridMate::GridSession* session) override {};
    void OnSessionError(GridMate::GridSession* session, const GridMate::string&
 error) override {};
    void OnMemberJoined(GridMate::GridSession* session, GridMate::GridMember*
member) {};
    void OnMemberLeaving(GridMate::GridSession* session, GridMate::GridMember*
 member) {};

    void Init() override { }
    void Activate() override
    {
       // The GridMate specific memory allocators must be created before setting
 up GridMate.
        // If multiple GridMate instances are to be created (e.g. one per com
ponent) this needs
        // to be called at that time that the ComponentApplication is initial
ized.
        AZ::AllocatorInstance<GridMate::GridMateAllocator>::Create();
        AZ::AllocatorInstance<GridMate::GridMateAllocatorMP>::Create();

        // The port that all game data will be transmitted.
        static const AZ::s32 GAME_SEARCH_PORT = 40000;
        // The port that is listening for clients searching for games on the
LAN.
        static const AZ::s32 GAME_PORT = 20000;

        // Initialize GridMate and services systems.
        mGridMate = GridMate::GridMateCreate(GridMate::GridMateDesc()   );
        mGridMate->StartOnlineService(GridMate::ST_LAN, GridMate::OnlineServi
ceDesc());
       mGridMate->StartMultiplayerService(GridMate::ST_LAN, GridMate::Session
ServiceDesc());

        // Attach this component to the event bus.
        AZ::TickBus::Handler::BusConnect();
        GridMate::SessionEventBus::Handler::BusConnect(mGridMate);

        // Configure the session
        GridMate::LANSessionParams sessionParam;
        sessionParam.m_topology = GridMate::ST_CLIENT_SERVER;
        sessionParam.m_numPublicSlots = 2;
        sessionParam.m_numPrivateSlots = 0;
        sessionParam.m_port = GAME_SEARCH_PORT;
        sessionParam.m_peerToPeerTimeout = 60000;
        sessionParam.m_flags = 0;
        sessionParam.m_numParams = 0;
        sessionParam.m_localMember = mGridMate->GetOnlineService()->GetUser();


        // Configure the carrier used by the host.
        GridMate::CarrierDesc carrierDesc;
        carrierDesc.m_enableDisconnectDetection = true;
        carrierDesc.m_connectionTimeoutMS = 10000;
        carrierDesc.m_threadUpdateTimeMS = 30;
        carrierDesc.m_version = 1;
        carrierDesc.m_port = GAME_PORT;

        // Start creating the session.
        // This operation is asynchronous and is only considered complete when

        // OnSessionCreated() or OnSessionError() events are called.
        mSession = mGridMate->HostSession(&params, carrierDesc);
    }
    void Deactivate() override
    {
        if(mSession)
            mSession->Leave(0);
        GridMate:SessionEventBus::Handler::BusDisconnect(mGridMate);
        AZ::TickBus::Handler::BusDisconnect();
        mGridMate->StopOnlineService();
        mGridMate->StopMultiplayerService();
        mGridMate = nullptr;
    }
    void OnTick(float dateTime, AZ::ScriptTimePoint t)
    {
        // Update the GridMate system each tick.
        mGridMate->Update();
    }

    static void Reflect(AZ::ReflectContext* context) {};
};
```

Searching for Sessions

To search for available sessions, call `SessionService::StartGridSearch()`. This function returns a `GridSearch` pointer, which on search completion contains a list of found sessions. Search completion events are sent via `GridSessionCallbacks::OnGridSearchComplete()` with the `GridSearch` pointer returned by `SessionService::StartGridSearch()`.

To cancel a search, use `GridSearch::AbortSearch()`.

The `GridSearch` pointer remains valid until the `GridSessionCallbacks::OnGridSearchRelease()` event occurs.

Joining a Session

Sessions can be joined by calling `SessionService::JoinSession()` with a `SearchInfo` pointer returned by a search. The function returns a `GridSession` pointer. When a client has successfully joined a session, an event is sent via `GridSessionCallbacks::OnSessionJoined()`.

The pointer returned by `SessionService::JoinSession()` remains valid until the `GridSessionCallbacks::OnSessionDelete()` event occurs.

Join session example

The following example component searches for any GridMate session and joins the first one that it finds.

```
// GridMateJoin
#include <AZCore/Component/Component.h>
#include <AZCore/Component/TickBus.h>
#include <GridMate/GridMate.h>
#include <GridMate/Session/LANSession.h>
#include <GridMate/Session/Session.h>

class JoinComponent
    : public AZ::Component
    , public AZ::TickBus::Handler
    , public GridMate::SessionEventBus::Handler
{
    GridMate::IGridMate* mGridMate;
    GridMate::GridSession* mSession;
    GridMate::GridSearch* mSearch;
public:
    AZ_COMPONENT(HostComponent, "{41900198-1122-4392-A579-CDDC85A23277}");

    HostComponent() : mSession(nullptr), mGridMate(nullptr) {}

    // Events from GridMate::SessionEventBus
    void OnSessionJoined(GridMate::GridSession* session) override {};
   void OnSessionError(GridMate::GridSession* session, const GridMate::string&
 error) override {};
    void OnMemberJoined(GridMate::GridSession* session, GridMate::GridMember*
member) {};
    void OnMemberLeaving(GridMate::GridSession* session, GridMate::GridMember*
 member) {};

    void OnGridSearchComplete(GridMate::GridSearch* search)
    {
        if(search->GetNumResults() > 0)
        {
            const GridMate::SearchInfo* result = search->GetResult(0);

            GridMate::CarrierDesc;
            carrierDesc.m_enableDisconnectDetection = true;
            carrierDesc.m_connectionTimeoutMS = 10000;
            carrierDesc.m_threadUpdateTimeMS = 30;
            carrierDesc.m_version = 1;
            carrierDesc.m_port = 0;

            // Join the session.
            // This operation is asynchronous and is only considered complete
when
            // OnSessionJoined() or OnSessionError() events are called.
            mSession = mGridMate->JoinSession(result, GridMate::JoinParams(),
carrierDesc);
        }
    }

    void Init() override { }
    void Activate() override
    {
       // The GridMate specific memory allocators must be created before setting
 up GridMate.
        // If multiple GridMate instances are to be created (e.g. one per com
ponent) this needs
        // to be called at that time that the ComponentApplication is initial
ized.
        AZ::AllocatorInstance<GridMate::GridMateAllocator>::Create();
        AZ::AllocatorInstance<GridMate::GridMateAllocatorMP>::Create();

        // The port that all game data will be transmitted.
        static const AZ::s32 GAME_SEARCH_PORT = 40000;
        // The port that is listening for clients searching for games on the
LAN.
        static const AZ::s32 GAME_PORT = 20000;

        // Initialize GridMate and services systems.
        mGridMate = GridMate::GridMateCreate(GridMate::GridMateDesc()   );
        mGridMate->StartOnlineService(GridMate::ST_LAN, GridMate::OnlineServi
ceDesc());
       mGridMate->StartMultiplayerService(GridMate::ST_LAN, GridMate::Session
ServiceDesc());

        // Attach this component to the event bus.
        AZ::TickBus::Handler::BusConnect();
        GridMate::SessionEventBus::Handler::BusConnect(mGridMate);

        // Search for games on the LAN.
        GridMate::LANSearchParams params;
        params.m_serverPort = GAME_SEARCH_PORT;
        params.m_localMember = mGridMate->GetOnlineService()->GetUser();
        mSearch = mGridMate->StartGridSearch(&params);
    }
    void Deactivate() override
    {
        if(mSession)
```

```
            mSession->Leave(0);
        GridMate:SessionEventBus::Handler::BusDisconnect(mGridMate);
        AZ::TickBus::Handler::BusDisconnect();
        mGridMate->StopOnlineService();
        GridMate::GridMateDestroy(mGridMate);
        mGridMate = nullptr;
    }

    void OnTick(float dateTime, AZ::ScriptTimePoint t)
    {
        // Update the GridMate system each tick.
        mGridMate->Update();
    }

    static void Reflect(AZ::ReflectContext* context) {};
};
```

Session Events

GridMate session events are sent via `GridMate::SessionEventBus`. To listen for session events, perform the following steps.

To listen for session events

1. Derive a listener from `SessionEventBus::Listener`.
2. Implement the `GridSessionCallbacks` interface.
3. Use the corresponding `GridMate` pointer to connect the listener.
4. To distinguish between the `GridSession` and `GridSearch` instances, pass the `GridSession` or `GridSearch` pointer together with the event.

Event descriptions

A description of each session event follows.

`virtual void OnSessionServiceReady()`
Callback that occurs when the session service is ready to process sessions.

`virtual void OnGridSearchStart(GridSearch* gridSearch)`
Callback when a grid search begins.

`virtual void OnGridSearchComplete(GridSearch* gridSearch)`
Callback that notifies the title when a game search query is complete.

`virtual void OnGridSearchRelease(GridSearch* gridSearch)`
Callback when a grid search is released (deleted). It is not safe to hold the grid pointer after this event.

`virtual void OnMemberJoined(GridSession* session, GridMember* member)`
Callback that notifies the title when a new member joins the game session.

`virtual void OnMemberLeaving(GridSession* session, GridMember* member)`
Callback that notifies the title that a member is leaving the game session.

Caution
The member pointer is not valid after the callback returns.

`virtual void OnMemberKicked(GridSession* session, GridMember* member)`
Callback that occurs when a host decides to kick a member. An `OnMemberLeaving` event is triggered when the actual member leaves the session.

`virtual void OnSessionCreated(GridSession* session)`
Callback that occurs when a session is created. After this callback it is safe to access session features. The host session is fully operational if client waits for the `OnSessionJoined` event.

`virtual void OnSessionJoined(GridSession* session)`
Called on client machines to indicate that the session has been joined successfully.

`virtual void OnSessionDelete(GridSession* session)`
Callback that notifies the title when a session will be left.

> **Caution**
> The session pointer is not valid after the callback returns.

`virtual void OnSessionError(GridSession* session, const string& errorMsg )`
Called when a session error occurs.

`virtual void OnSessionStart(GridSession* session)`
Called when the game (match) starts.

`virtual void OnSessionEnd(GridSession* session)`
Called when the game (match) ends.

`virtual void OnMigrationStart(GridSession* session)`
Called when a host migration begins.

`virtual void OnMigrationElectHost(GridSession* session,GridMember*& newHost)`
Called to enable the user to select a member to be the new Host.

> **Note**
> The value is ignored if it is null, if the value is the current host, or if the member has an invalid connection ID.

`virtual void OnMigrationEnd(GridSession* session,GridMember* newHost)`
Called when the host migration is complete.

`virtual void OnWriteStatistics(GridSession* session, GridMember* member, StatisticsData& data)`
Called at the last opportunity to write statistics data for a member in the session.

Physics

This section describes the Physics system and how to interact with the physics engine.

To create a physical world object, you use the `CreatePhysicalWorld()` function. The `CreatePhysicalWorld()` function returns a pointer to the `IPhysicalWorld` interface. You then fill the world with geometries and physical entities. You control the entities with the functions described in this section. Some functions apply to almost all entities, while others apply to specific entity structures. Other functions control how entities interact and how the world affects them.

The following sections describe these topics in detail:

Topics

Geometries

Geometries are first created as independent objects so that they can be used alone via the `IGeometry` interface which they expose and then they can be physicalized and added to physical entities. Geometry physicalization means computing physical properties (volume, inertia tensor) and storing them in a special internal structure. Pointers to these structures can then be passed to physical entities.

Each physicalized geometry has a reference counter which is set to 1 during creation, incremented every time the geometry is added to an entity and decremented every time the geometry is removed or the entity is deleted. When the counter reaches 0, the physical geometry structure is deleted and the corresponding `IGeometry` object is released. The `IGeometry` object is also reference counted.

Geometry Management Functions

Geometry management functions are accessible through the geometry manager, which is a part of physical world. To obtain a pointer to the geometry manger, call the `GetGeomManager()` function.

CreateMesh

The `CreateMesh` geometry management function creates a triangular mesh object from a set of vertices and indices (3 indices per triangle) and returns the corresponding IGeometry pointer.

- The engine uses triangle connectivity information in many places, so it is strongly recommended to have meshes closed and manifold. The function is able to recognize different vertices that represent the same point in space for connectivity calculations (there is no tolerance though, it checks only for exact duplicates). Open edges are ok only for geometries that will not be used as parts of dynamic physical entities and only if there will be little or no interaction with them.
- For collision detection the function can create either an OBB or a memory-optimized AABB or a single box tree. Selection is made by specifying the corresponding flag. If both AABB and OBB flags are specified, the function selects the tree that fits the geometry most tightly. Since an OBB tree is tighter in most cases, priority of AABBs can be boosted to save memory (also, AABB checks are slightly faster if the trees are equally tight). The engine can either copy the vertex/index data or use it directly from the pointers provided.
- The `mesh_multycontact flags` give some hints on whether multiple contacts are possible. Specifying that multiple contacts are unlikely (`mesh_multycontact0`) can improve performance a bit at the expense of missing multiple contacts if they do occur (note that it does not necessarily mean they will be missed, it is a hint for the algorithm to use some optimizations more aggressively). `mesh_multycontact2` disables this optimization and ..1 is a recommended default setting. Convex geometries are detected and some additional optimizations are used for them, although internally there is no separate class for convex objects (this may change in the future).
- Meshes can have per-face materials. Materials are used to look up friction, bounciness, and pierceability coefficients and can be queried by the game as a part of collision detection output.
- The `CreateMesh` function is able to detect meshes that represent primitives (with the specified tolerance) and returns primitive objects instead. In order to activate this detection, the corresponding flags should be specified. Note that primitives can't store materials. They can only have one in the physical geometry structure, so this detection is not used when the material array has more than one material index in it.

CreatePrimitive

`CreatePrimitive:` Creates a primitive geometry explicitly. The corresponding primitive (cylinder, sphere, box, heightfield, or ray) structure should be filled and passed as a parameter, along with its ::type.

RegisterGeometry

`RegisterGeometry` physicalizes an `IGeometry` object by computing its physical properties and storing them in an auxiliary structure. Material index (`surfaceidx`) can be stored in it; it will be used if the geometry itself does not have any materials specified (such as if it is a primitive). `AddRefGeometry` and `UnregisterGeometry` comprise a reference "sandwich" for it. Note that the latter does not delete the object until its reference count becomes 0.

Geometries and physicalized geometries can be serialized. This saves time when computing OBB trees. That computation is not particularly slow, but serialization is faster.

Physical Entities

Physical entities can be created via calls to the `CreatePhysicalEntity` method of the physical world. `CreatePhysicalEntity` can create the types of entities noted in the following table:

Physical Entity Types

Type	Description
PE_STATIC	An immovable entity. An immovable entity can still be moved manually by setting positions from outside, but in order to ensure proper interactions with simulated objects, it is better to use PE_RIGID entity with infinite mass.
PE_RIGID	A single rigid body. Can have infinite mass (specified by setting mass to 0), in which case it will not be simulated but will interact properly with other simulated objects;
PE_ARTICU-LATED	An articulated structure, consisting of several rigid bodies connected with joints (a ragdoll, for instance). It is also possible to manually connect several PE_RIGID entities with joints but in this case they will not know that they comprise a single object, and thus some useful optimizations will not be used.
PE_WHEELED-VEHICLE	A wheeled vehicle. Internally it is built on top of a rigid body, with added vehicle functionality (wheels, suspensions, engine, brakes). PE_RIGID, PE_ARTICULATED and PE_WHEELEDVEHICLE are purely physical entities that comprise the core of the simulation engine. The other entities are processed independently.
PE_LIVING	A special entity type to represent player characters that can move through the physical world and interact with it.
PE_PARTICLE	A simple entity that represents a small lightweight rigid body. It is simulated as a point with some thickness and supports flying, sliding and rolling modes. Recommended usage: rockets, grenades and small debris.
PE_ROPE	A rope object. It can either hang freely or connect two purely physical entities.
PE_SOFT	A system of non-rigidly connected vertices that can interact with the environment. A typical usage is cloth objects.

Creating and managing entities

When creating and managing entities, keep in mind the following:

- Entities use a two-dimensional, regular grid to speed up broad phase collision detection. The grid should call the `SetupEntityGrid` function before physical entities are created.
- Entities can be created in permanent or on-demand mode and are specified by the parameter `lifeTime` (use 0 for permanent entities). For on-demand mode, the entity placeholders should be created first using `CreatePhysicalPlaceholder`. Physics will then call the outer system to create the full entity whenever an interaction is required in the bounding box for this placeholder.
- If an entity is not involved in any interactions for the specified lifetime, it will be destroyed, with the placeholder remaining. Placeholders require less memory than full entities (around 70 bytes versus 260 bytes). It is possible for an outer system to support hierarchical placeholders, such as meta-placeholders that create other placeholders upon request.
- A sector-based, on-demand physicalization is activated after `RegisterBBoxInPODGrid` is called. Entities are created and destroyed on a sector basis. The sector size is specified in `SetupEntityGrid`.
- You can use `SetHeightfieldData` to set up one special static terrain object in the physical world. You can also create unlimited terrain geometry manually and add it to an entity.

Destroying, suspending, and restoring entities

To destroy, suspend, or restore a physical entity, use `DestroyPhysicalEntity` and set the `mode` parameter to 0, 1, or 2, respectively. Suspending an entity clears all of its connections to other entities, including constraints, without actually deleting the entity. Restoring an entity after suspension will not restore all lost connections automatically. Deleted entities are not destroyed immediately; instead, they are put into a recycle bin. You might need to remove references to any one-way connections. The recycle bin is emptied at the end of each `TimeStep`. You can also call `PurgeDeletedEntities`.

Physical entity IDs

All physical entities have unique IDs that the physics engine generates automatically. You do not need to specify an ID during creation. You can also set a new ID later. Entities use these IDs during serialization to save dependency information. When reading the saved state, be sure that entities have the same IDs. IDs are mapped to entity pointers by use of an array, so using large ID numbers will result in allocation of an equally large array.

Associations with outside objects

To maintain associations with outside engine objects, physical entities store an additional void pointer and two 16-bit integers (`pForeignData`, `iForeignData`, and `iForeignFlags`) . These parameters are set from outside, not by the entities. Use `pForeignData` to store a pointer to the outside engine reference entity and `iForeignData` to store the entity type, if applicable.

For each material index, the physical world stores the friction coefficient, a bounciness (restitution) coefficient, and flags. When two surfaces contact, the contact's friction and bounciness are computed as an average of the values of both surfaces. The flags only affect raytracing.

Simulation type

Physical entities are grouped by their simulation type, in order of increasing "awareness". Certain interface functions, such as ray tracing and querying entities in an area, allow filtering of these entities by type.

- 0 (`bitmask ent_static`) – Static entities. Although terrain is considered static, it does not have a special simulation type. It can be filtered independently with the `ent_terrain` bitmask.
- 1 (`bitmask ent_sleeping_rigid`) – Deactivated, physical objects (rigid bodies and articulated bodies).
- 2 (`bitmask ent_rigid`) – Active, physical objects.
- 3 (`bitmask ent_living`) – Living entities.
- 4 (`bitmask ent_independent`) – Physical entities that are simulated independently from other entities (particles, ropes, and soft objects).
- 6 (`bitmask ent_triggers`) – Entities (or placeholders) that are not simulated and only issue callbacks when other entities enter their bounding box.
- 7 (`bitmask ent_deleted`) – Objects in the recycle bin. Do not use this directly.

Entities that have a lower simulation type are not aware of entities with higher simulation types (types 1 and 2 are considered as one for this purpose), so players (type 3) and particles (type 4) check collisions against physical entities (types 1 and 2) but physical entities do not know anything about them. Similarly, ropes (type 4) can check collisions against players but not the other way. However, entities of higher types can still affect entities with lower types by using impulses and constraints. Most entities expect a particular simulation type (and will automatically set to the proper value).

There are exceptions to the 'awareness hierarchy': for example, articulated entities can be simulated in types 1 and 2 as fully physicalized dead bodies, or in type 4 as skeletons that play impact animations without affecting the environment and being affected by it.

Functions for Physical Entities

Most interactions with physical entities will use the functions `AddGeometry`, `SetParams`, `GetParams`, `GetStatus`, and `Action`.

- `AddGeometry` – Adds multiple geometries (physicalized geometries) to entities. For more details, see the AddGeometry section that follows.
- `RemoveGeometry` – Removes geometries from entities.
- `SetParams` – Sets parameters.
- `GetParams` – Gets the simulation input parameters.
- `GetStatus` – Gets the simulation output parameters. `GetStatus` requests the values that an entity changes during simulation.
- `Action` – Makes an entity execute an action, such as adding an impulse.

These functions take structure pointers as parameters. When you want to issue a command, you can create a corresponding structure (for example, as a local variable) and specify only the fields you need. The constructor of each structure provides a special value for all fields that tells the physics engine that the field is unused. You can also do this explicitly by using the `MARK_UNUSED` macro and `is_unused` to verify that the field is unused.

AddGeometry

`AddGeometry` adds a physicalized geometry to an entity. Each geometry has the following properties:

- `id` – A unique part identifier within the bounds of the entity to which the geometry belongs. You can specify the ID or use `AddGeometry` to generate a value automatically. The ID doesn't change if the parts array changes (for example, if some parts from the middle are removed), but the internal parts index might change.
- `position`, `orientation`, and `uniform scaling` – Relative to the entity.
- `mass` – Used for non-static objects; static objects assume infinite mass in all interactions. You can specify the mass or density where the complementary value will be computed automatically (using formula mass = density*volume; `volume` is stored in the physicalized geometry structure and scaled if the geometry is scaled).
- `surface_idx` – Used if neither IGeometry nor physicalized geometry have surface (material) identifiers.
- `flags` and `flagsCollider` – When an entity checks collisions against other objects, it checks only parts that have a flag mask that intersects its current part's `flagsCollider`. You can use 16-type bits (`geom_colltype`) to represent certain entity groups. Although not enforced, it is good practice to keep these relationships symmetrical. If collision checks are known to be one-sided (for example, entity A can check collisions against entity B but never in reverse), you can choose to not maintain this rule. Certain flags are reserved for special collision groups, such as `geom_colltype1 = geom_colltype_players` and `geom_colltype2 = geom_colltype_explosion` (when explosion pressure is calculated, only parts with this flag are considered). There are also special flags for raytracing and buoyancy calculations: `geom_colltype_ray` and `geom_floats`.
- `minContactDist` – The minimum distance between contacts the current part of the entity might have with another part of an entity. Contacts belonging to different parts are not checked for this. You can leave this unused so it will initialize with a default value based on geometry size. Each part can have both geometry and proxy geometry. Geometry is used exclusively for raytracing and proxy geometry. If no proxy geometry is specified, both geometries are set to be equal to allow the raytracing to test against high-poly meshes without needing to introduce changes to the part array layout.

Functions for Entity Structures

This section describes functions that control general and specific kinds of entity structures.

Topics

Common Functions

pe_params_pos

Sets the position and orientation of the entity. You can use offset/quaternion/scaling values directly or allow the physics to extract them from a 3x3 (orientation+scaling) or a 4x4 (orientation_scaling+offset) matrix. Physics use a right-to-left transformation order convention, with vectors being columns (`vector_in_world = Matrix_Entity * Matrix_Entity_Parts * vector_in_geometry`). All interface structures that support matrices can use either row-major or column-major matrix layout in memory (the latter is considered transposed; thus, the corresponding member has T at the end of its name).

There is no per-entity scaling; scaling is only present for parts. When a new scaling is set with `pe_params_pos`, it is copied into each part and overrides any previous individual scalings. This structure also allows you to set the simulation type manually. After changes are made, entity bounds are typically recalculated and the entity is re-registered in the collision hash grid; however, this can be postponed if `bRecalcBounds` is set to `0`.

pe_params_bbox

Sets an entity's bounding box to a particular value, or queries it when used with `GetParams`). The bounding box is recalculated automatically based on the entity's geometries, but you can set the bounding box manually for entities without geometries (for example, triggers) or placeholders. If the entity has geometries, it might recalculate its bounding box later, overriding these values. Bounding boxes are axis-aligned and in the world coordinate system.

pe_params_outer_entity

Specifies an outer entity for an entity. When a box of interest (its center) is inside the entity with an outer entity, the outer entity is excluded from the set of potential colliders. This allows you to have a building exterior quickly culled away when the region of interest is inside the building's interior. Outer entities can be nested and an optional geometry to test for containment is supported.

pe_params_part

Sets or queries the entity part's properties. The part can be specified using an internal part index or its ID.

pe_simulation_params

Sets simulation parameters for entities that can accept these parameters (e.g. physical entities, ropes, and soft entities). `minEnergy` is equal to sleep speed squared. Damping and gravity can be specified independently for colliding and falling state, for example when there are no contacts.

pe_params_buoyancy

Sets the buoyancy properties of the object and the water plane. The physics engine does not have a list of water volumes, so the outer system must update water plane parameters when they change. The water surface is assumed to be a perfect plane, so you can simulate bobbing of the waves by disturbing the normal of this surface. `waterFlow` specifies the water movement velocity and affects the object based on its `waterResistance` property). A separate sleeping condition is used in the water (`waterEmin`).

pe_params_sensors

Attaches sensors to entities. Sensors are rays that the entity can shoot to sample the environment around it. It is more efficient to do it from inside the entity step than by calling the world's raytracing function for every ray from outside the entity step. Living entities support vertical-down sensors.

pe_action_impulse

Adds a one-time impulse to an entity. `impulse` is the impulse property (in N*sec; impulse P will change the object's velocity by P/[object mass]). `point` is a point in world space where the impulse is applied and used to calculate the rotational effects of the impulse. The `of point` momentum can be used to specify the rotational impulse explicitly. If neither the point nor momentum are specified, the impulse is applied to the center of the mass of the object. `iApplyTime` specifies the time when the impulse is applied. By default the value is 2 ("after the next step") to allow the solver an opportunity to reflect the impulse.

pe_action_add_constraint

Adds a constraint between two objects. Points specify the constraint positions in world space. If the second point is used and different from the first point, the solver will attempt to join them.

Relative positions are always fully constrained to be 0 (i.e. the points on the bodies will always be in one spot) and relative rotations can be constrained in twist and bend directions. These directions correspond to rotation around the x-axis and the remaining rotation around a line on the yz-plane (tilt of the x axis) of a relative transformation between the two constraint coordinate frames attached to the affected bodies.

The original position of the constraint frames are specified with `qframe` parameters in world or entity coordinate space (as indicated by the corresponding flag in `flags`). If one or both qframes are unused, they are considered to be an identity transformation in either the world or entity frame.

Rotation limits are specified with the `xlimits` and `yzlimits` parameters, with valid element values of 0 (minimum) and 1 (maximum). If the minimum is more than or equal to the maximum, the corresponding relative rotation is prohibited. `pConstraintEntity` specifies an entity that represents the constraint. When passed a pe_action_add_constraint pointer, `Action` returns a constraint identifier that can be used to remove the constraint. 0 indicates a failure.

pe_action_set_velocity

Sets the velocity of an object, which is useful for rigid bodies with infinite mass (represented as mass). `pe_action_set_velocity` informs the physics system about the body's velocity, which can help the solver ensure zero relative velocity with the objects contacted. If velocity is not set and only the position is changed, the engine relies solely on penetrations to enforce the contacts. Velocity will not be computed automatically if the position is set manually each frame. The body will continue moving with the specified velocity once it has been set.

pe_status_pos

Requests the current transformation (position, orientation, and scale) of an entity or its part. You can also use `pe_params_pos` with `GetParams`. If matrix pointers are set, the engine will provide data in the corresponding format. The `BBox` member in this structure is relative to the entity's position.

pe_status_dymamics

Retrieves an entity's movement parameters. Acceleration and angular acceleration are computed based on gravity and interactions with other objects. External impulses that might have been added to the entity are considered instantaneous. `submergedFraction` is a fraction of the entity's volume under water during the last frame (only parts with the `geom_float` flag are considered). `waterResistance` contains the maximum water resistance that the entity encountered in one frame since the status was last requested (the accumulated value is cleared when the status is returned). This value can be useful for generating splash effects.

Living Entity-Specific Functions

Living entities use cylinders or capsules as their bounding geometry. Normally the cylinders are hovering above the ground and the entity shoots a single ray down to detect if it is standing on something. This cylinder geometry always occupies the first part slot (it is created automatically). It is possible to add more geometries manually, but they will not be tested against the environment when the entity moves. However, other entities will process them when testing collisions against the entity.

Living entities never change their orientation themselves; this is always set from outside. Normally, living entities are expected to rotate only around the z-axis, but other orientations are supported. However, collisions against living entities always assume vertically oriented cylinders.

pe_player_dimensions (GetParams | SetParams)

Sets the dimensions of the living entity's bounding geometry.

`heightPivot` specifies the z-coordinate of a point in the entity frame that is considered to be at the feet level (usually 0).

`heightEye` is the z-coordinate of the camera attached to the entity. This camera does not affect entity movement, its sole purpose is to smooth out height changes that the entity undergoes (during walking on a highly bumpy surface, such as stairs, after dimensions change and during landing after a period of flying). The camera position can be requested via the `pe_status_living` structure.

`sizeCollider` specifies the size of the cylinder (x is radius, z is half-height, y is unused).

`heightColliders` is the cylinder's center z-coordinate.

The head is an auxiliary sphere that is checked for collisions with objects above the cylinder. Head collisions don't affect movement but they make the camera position go down. `headRadius` is the radius of this sphere and `headHeight` is the z-coordinate of its center in the topmost state (that is, when it doesn't touch anything).

pe_player_dynamics (GetParams | SetParams)

Sets a living entity's movement parameters. Living entities have their 'desired' (also called 'requested') movement velocity (set with `pe_action_move`) and they attempt to reach it. How fast that happens depends on the `kInertia` setting. The greater this value is, the faster the velocity specified by `pe_action_move` is reached. The default is 8. 0 means that the desired velocity will be reached instantly.

`kAirControl` (0..1) specifies how strongly the requested velocity affects movement when the entity is flying (1 means that whenever a new requested velocity is set, it is copied to the actual movement velocity).

`kAirResistance` describes how fast velocity is damped during flying.

`nodSpeed` (default 60) sets the strength of camera reaction to landings.

`bSwimming` is a flag that tells that the entity is allowed to attempt to move in all directions (gravity might still pull it down though). If not set, the requested velocity will always be projected on the ground if the entity is not flying.

`minSlideAngle`, `maxClimbAngle`, `maxJumpAngle` and `minFallAngle` are threshold angles for living entities that specify maximum or minimum ground slopes for certain activities. Note that if an entity's bounding cylinder collides with a sloped ground, the behavior is not governed by these slopes only.

Setting `bNetwork` makes the entity allocate a much longer movement history array which might be required for synchronization (if not set, this array will be allocated the first time network-related actions are requested, such as performing a step back).

Setting `bActive` to 0 puts the living entity to a special 'inactive' state where it does not check collisions with the environment and only moves with the requested velocity (other entities can still collide with it, though; note that this applies only to the entities of the same or higher simulation classes).

pe_action_move

Requests a movement from a living entity. `dir` is the requested velocity the entity will try to reach. If `iJump` is not 0, this velocity will not be projected on the ground, and snapping to the ground will be turned off for a short period of time. If `iJump` is 1, the movement velocity is set to be equal to `dir` instantly. If `iJump` is 2, `dir` is added to it. `dt` is reserved for internal use.

pe_status_living

Returns the status of a living entity.

`vel` is the velocity that is averaged from the entity's position change over several frames.

`velUnconstrained` is the current movement velocity. It can be different from `vel` because in many cases when the entity bumps into an obstacle, it will restrict the actual movement but keep the movement velocity the same, so that if on the next frame the obstacle ends, no speed will be lost.

`groundHeight` and `groundSlope` contain the point's `z` coordinate and normal if the entity is standing on something; otherwise, `bFlying` is 1. Note that `pGroundCollider` is set only if the entity is standing on a non-static object.

`camOffset` contains the current camera offset as a 3d vector in the entity frame (although only `z` coordinates actually changes in it).

`bOnStairs` is a heuristic flag that indicates that the entity assumes that it is currently walking on stairs because of often and abrupt height changes.

Particle Entity-Specific Functions

pe_params_particle

Sets particle entity parameters.

During movement, particles trace rays along their paths with the length `size*0.5` (since `size` stands for 'diameter' rather than 'radius') to check if they hit something. When they lie or slide, they position themselves at a distance of `thickness*0.5` from the surface (thus thin objects like shards of glass can be simulated).

Particles can be set to have additional acceleration due to thrust of a lifting force (assuming that they have wings) with the parameters `accThrust` and `accLift` but these should never be used without specifying `kAirResistance`; otherwise, particles gain infinite velocity.

Particles can optionally spin when in the air (toggled with flag `particle_no_spin`). Spinning is independent from linear motion of particles and is changed only after impacts or falling from surfaces.

Particles can align themselves with the direction of the movement (toggled with `particle_no_path_alignment` flag) which is very useful for objects like rockets. That way, the y-axis of the entity is aligned with the heading and the z-axis is set to be orthogonal to y and to point upward ('up' direction is considered to be opposite to particle's gravity).

When moving along a surface, particles can either slide or roll. Rolling can be disabled with the flag `particle_no_roll` (it is automatically disabled on steep slopes). Note that rolling uses the particle material's friction as damping while rolling treats friction in a conventional way. When touching ground, particles align themselves so that their normal (defined in entity frame) is parallel to the surface normal.

Particles can always keep the initial orientation as well (`particle_constant_orientation`) and stop completely after the first contact (`particle_single_contact`). `minBounceVel` specifies the lower velocity threshold after which the particle will not bounce, even if the bounciness of the contact is more than 0.

Articulated Entity-Specific Functions

Articulated entities consist of linked, rigid bodies called structural units. Each structural unit has a joint that connects it to its parent. For the connection structure, you should use a tree with a single root. Linked loops are not allowed.

Articulated entities can simulate body effects without interactions with the environment by using featherstone mode, which you can tweak so that the entity tolerates strong impacts and so that complex body structures have stiff springs. Articulated entities use a common solver for interactive mode.

pe_params_joint

You can use `pe_params_joint` to:

- Create a joint between two bodies in an articulated entity
- Change the parameters of an existing articulated entity
- Query the parameters of an existing articulated entity, when used with `GetParams`

A joint is created between the two bodies specified in the `op` parameter at the pivot point (in the entity frame). When a geometry is added to an articulated entity, it uses `pe_articgeomparams` to specify which body the geometry belongs to (in `idbody`). `idbody` can be any unique number and each body can have several geometries. There are no restrictions on the order in which joints are created, but all bodies in an entity must be connected before the simulation starts.

Joints use Euler angles to define rotational limits. Flags that start with `angle0_` can be specified individually for each angle by shifting left by the 0-based angle index. For example, to lock the z-axis you can use OR the flags with `angle0_locked`<<2). The child body inherits the coordinate frame from the first entity (geometry) that was assigned to it.

Joint angular limits are defined in a relative frame between the bodies that the joint connects. Optionally the frame of the child body can be offset by specifying a child's orientation that corresponds to rotation angles (0,0,0), using q0, pMtx0, or pMtx0T. This can help to get limits that can be robustly represented using Euler angles.

A general rule for limits is to set upper and lower bounds at least 15 to 20 degrees apart (depending on simulation settings and the height of the joint's velocity) and to keep the y-axis limit in the -90..90 degrees range (preferably within safe margins from its ends).

Note
All angles are defined in radians in the parameter structure.

`pe_params_joint` uses 3D vectors to represent groups of three values that define properties for each angle. In addition to limits, each angle can have a spring that will pull the angle to 0 and a dashpot that will dampen the movement as the angle approaches its limit. Springs are specified in acceleration terms: stiffness and damping can stay the same for joints that connect bodies with different masses, and damping can be computed automatically to yield a critically damped spring by specifying `auto_kd` for the corresponding angle.

`joint_no_gravity` makes the joint unaffected by gravity, which is useful if you assume forces that hold the joint in its default position are enough to counter gravity). This flag is supported in featherstone mode.

`joint_isolated_accelerations` makes the joint use a special mode that treats springs like guidelines for acceleration, which is recommended for simulating effects on a skeleton. This flag is supported in featherstone mode.

Effective joint angles are always the sum of `q` and `qext`. If springs are activated, they attempt to drive `q` to 0. The allows you to set a pose from animation and then apply physical effects relative to it. In articulated entities, collisions are only checked for pairs that are explicitly specified in `pSelfCollidingParts` (this setting is per body or per joint, rather than per part).

pe_params_articulated_body

`pe_params_articulated_body` allows you to set and query articulated entity simulation mode parameters. Articulated entities can be attached to something or be free, and are set by the `bGrounded` flag. When grounded, the entity can:

- Fetch dynamic parameters from the entity it is attached to (if `bInreritVel` is set; the entity is specified in `pHost`)
- Be set using the `a`, `wa`, `w` and `v` parameters

`bCollisionResp` switches between featherstone mode (0) and constraint mode (1).

`bCheckCollisions` turns collision detection on and off. It is supported in constraint mode.

`iSimType` specifies a simulation type, which defines the way in which bodies that comprise the entity evolve. Valid values:

- `0` – joint pivots are enforced by projecting the movement of child bodies to a set of constrained directions |
- `1` – bodies evolve independently and rely on the solver to enforce the joints. The second mode is not supported in featherstone mode. In constraint mode, it is turned on automatically if bodies are moving fast enough.

  We recommend setting this value to `1` to make slow motion smoother.

Lying mode

Articulated entities support a lying mode that is enabled when the number of contacts is greater than a specified threshold (`nCollLyingMode`). Lying mode has a separate set of simulation parameters, such as gravity and damping. This feature was designed for ragdolls to help simulate the high damping of a human body in a simple way, for example by setting gravity to a lower value and damping to a higher than usual value.

Standard simulation versus freefall parameters

Standard simulation parameters can be different from freefall parameters. When using the constraint mode, articulated entities can attempt to represent hinge joints (rotational joints with only axis enabled) as two point-to-point constraints by setting the `bExpandHinges` parameter (this value propagates to `joint_expand_hinge` flags for all joints, so you do not need to manually set the value for joints).

Rope Entity-Specific Functions

Ropes are simulated as chains of connected equal-length sticks ("segments") with point masses. Each segment can individually collide with the environment. Ropes can tie two entities together. In this case ropes add a constraint to the entities when the ropes are fully strained and won't affect their movement.

In order to collide with other objects (pushing them if necessary) in a strained state, the rope must use dynamic subdivision mode (set by `rope_subdivide_segs` flag).

pe_params_rope

Specifies all the parameters a rope needs to be functional.

Rope entities do not require any geometry. If you do not specify initial point positions, the rope is assumed to be hanging down from its entity position. If you do specify initial point positions, segments should have equal length but within some error margin. Ropes use an explicit friction value (not materials) to specify friction.

If `pe_params_rope` is passed to `GetParams`, `pPoints` will be a pointer to the first vertex in an internal rope vertex structure, and `iStride` will contain the size of it.

Soft Entity-Specific Functions

There are two types of soft entities: *mesh-based* and *tetrahedral lattice-based*. Mesh based entities use a soft, constraint-like solver and are typically cloth objects. Tetrahedral lattice-based entities use a spring solver and are typically jelly-like objects.

The longest edges of all triangles can optionally be discarded with the `sef_skip_longest_edges` flag.

Collisions are handled at the vertex level only (although vertices have a customizable thickness) and work best against primitive geometries rather than meshes.

pe_params_softbody

This is the main structure to set up a working soft entity (another one is `pe_simulation_params`).

Thickness

The thickness of the soft entity is the collision size of vertices (they are therefore treated as spheres). If an edge differs from the original length by more than `maxSafeStep`, positional length enforcement occurs.

Damping

Spring damping is defined with `kdRatio` as a ratio to a critically damped value (overall damping from `pe_simulation_params` is also supported).

Wind

Soft entities react to wind if `airResistance` is not 0 (if wind is 0, having non-zero `airResistance` would mean that the entity will look like it is additionally damped - air resistance will attempt to even surface velocity with air velocity).

Water

Soft entities react to water in the same way that they react to wind, but the parameters specified in pe_params_buoyancy are used instead. Note that the Archimedean force that acts on vertices submerged in the water will depend on the entity's density which should be defined explicitly in pe_simulation_params (dependence will be same as for rigid bodies - the force will be 0 if waterDensity is equal to density). collTypes enables collisions with entities of a particular simulation type using ent_ masks.

pe_action_attach_points

Can be used to attach some of a soft entity's vertices to another physical entity.

piVtx specifies vertex indices.

points specify attachment positions in world space. If points values are not specified, current vertex positions are the attachment points.

Collision Classes

Use collision classes to filter collisions between two physical entities. A collision class comprises two 32-bit uints, a type, and an ignore.

You can use collision classes to implement scenarios such as "player only collisions," which are objects passable by AI actors but not passable by players. This feature allows you to configure filtering of the collision between physical entities independently of their collision types.

Setup

Physical entities can have one or more collision classes and can ignore one or more collision classes. To have a physical entity ignore a collision, use the ignore_collision attribute of the <Physics> element in the <SurfaceType> definition, as shown in the following example:

SurfaceTypes.xml

```
<SurfaceType name="mat_nodraw_ai_passable">
  <Physics friction="0" elasticity="0"  pierceability="15" ignore_collision="col
lision_class_ai"/>
</SurfaceType>
```

All physical entity types such as LivingEntity and ParticleEntity are supplied with default collision classes like collision_class_living and collision_class_particle. Living entity uses one additional game specific collision class: either collision_class_ai for AI actors, or collision_class_player for players.

Player.lua

```
Player = {
...
  physicsParams =
  {
    collisionClass=collision_class_player,
  },
...
}
```

`BasicAI.lua`

```
BasicAI = {
...
  physicsParams =
  {
    collisionClass=collision_class_ai,
  },
...
}
```

Code

```
struct SCollisionClass
{
    uint32 type;     // collision_class flags to identify the entity
    uint32 ignore;   // another entity will be ignored if *any* of these bits
are set in its type
};
```

The `type` identifies which entity the collision classes belong to.

Some collision classes like the following are defined in `CryPhysics`:

- `collision_class_terrain`
- `collision_class_wheeled`
- `collision_class_living`
- `collision_class_articulated`
- `collision_class_soft`
- `collision_class_roped`
- `collision_class_particle`

Other collision classes are defined in `GamePhysicsSettings.h`, starting from the `collision_class_game` bit:

```
#define GAME_COLLISION_CLASSES(f) \
    f( gcc_player_capsule,     collision_class_game << 0) \
    f( gcc_player_body,        collision_class_game << 1) \
    f( gcc_pinger_capsule,     collision_class_game << 2) \
    f( gcc_pinger_body,        collision_class_game << 3) \
    f( gcc_vehicle,            collision_class_game << 4) \
    f( gcc_large_kickable,     collision_class_game << 5) \
    f( gcc_ragdoll,            collision_class_game << 6) \
    f( gcc_rigid,              collision_class_game << 7) \
    f( gcc_alien_drop_pod,     collision_class_game << 8) \
    f( gcc_vtol,               collision_class_game << 9) \
```

All these classes are automatically exposed to Lua. Brushes and most objects have the collision classes available in the properties through the editor.

Types

For types, you can set many or zero bits.

In the following example, of the classes `LIVING`, `PLAYER`, `TEAM1`, `TEAM2`, `AI`, `AI_1`, and `AI_2`, `player1` belongs to the `LIVING` entity class, the `PLAYER` class, and the `TEAM1` class:

```
SCollisionClass player1(0,0), player2(0,0), ai1(0,0), ai7(0,0), object1(0,0);

player1.type = LIVING|PLAYER|TEAM1;
player2.type = LIVING|PLAYER|TEAM2;
ai1.type = LIVING|AI|AI_1;
ai7.type = LIVING|AI|AI_2;
object1.type = 0;
```

Filtering the collision

Filtering occurs by checking the `type` of one entity against the `ignore` of another entity.

This is done both ways, and if bits overlap, then the collision is ignored. For example:

```
bool ignoreCollision = (A->type & B->ignore) || (A->ignore & B->type);
```

If you want `ai7` to ignore collisions with anything that has `AI_1` set, then add `AI_1` to the `ignore` flags like this:

```
ai7.ignore = AI_1
```

If you want `object1` to ignore all living physical entities, set its `ignore` flag like this:

```
object1.ignore=LIVING
```

Interface

- For code, see `physinterface.h` and `GamePhysicsSettings.h`.
- To access and set the collision classes on the physical entity, use `*pe_collision_class struct SCollisionClass pe_params_collision_class`.
- For helpers that set additional ignore maps, see `GamePhysicsSettings.h`.
- In Lua, see `SetupCollisionFiltering` and `ApplyCollisionFiltering`. Lua script-binding is done through `SetPhysicParams(PHYSICPARAM_COLLISION_CLASS)`.

Functions for World Entities

Use the functions in this section to modify entities or a physical world environment.

Advancing the Physical World Time State

The `TimeStep` functions make the entities advance their state by the specified time interval.

If `timeGranularity` in the physical variables is set, the time interval will be snapped to an integer value with the specified granularity (for example, if `timeGranularity` is 0.001, the time interval will be snapped to a millisecond).

Entities that perform the step can be filtered with `ent_` flags in the `flags` parameter.

The `flags` parameter can contain `ent_` masks for Simulation type (p. 354).

The `flags` parameter can also contain the `ent_flagged_only` flag. This flag causes entities to be updated only if the entities have the `pef_update` flag set.

Specifying the `ent_deleted` flag will allow the world to delete entities that have timed out if physics on demand is used.

Most entities have the maximum time step capped. To have larger timesteps, entities have to perform several substeps. The number of substeps can be limited with the physics variable `nMaxSubsteps`.

Returning Entities with Overlapping Bounding Boxes

The function `GetEntitiesInBox` uses the internal entity hash grid to return the number of entities whose bounding boxes overlap a specified box volume. The function supports filtering by Simulation type (p. 354) and optional sorting of the output list by entity mass in ascending order.

Syntax

```
virtual int GetEntitiesInBox(Vec3 ptmin,Vec3 ptmax, IPhysicalEntity **&pList,
int objtypes, int szListPrealloc=0) = 0;
```

Example call

```
IPhysicalEntity** entityList = 0;
int entityCount = gEnv->pPhysicalWorld->GetEntitiesInBox(m_volume.min,
m_volume.max, entityList,
   ent_static | ent_terrain | ent_sleeping_rigid | ent_rigid);
```

Parameters

Paramet-er	Description	
`ptmin`	Minimum point in the space that defines the desired box volume.	
`ptmax`	Maximum point in the space that defines the desired box volume.	
`pList`	Pointer to a list of objects that the function poplulates.	
`obj-types`	Types of objects that need to be considered in the query.	
`szList-Preal-loc`	If specified, the maximum number of objects contained in the `pList` array.	

The possible object types are described in the `physinterface.h` header file in the `entity_query_flags` enumerators. A few are listed in the following table:

Entity type flag	Description	
`ent_static`	Static entities	
`ent_terrain`	Terrain	
`ent_sleeping_rigid`	Sleeping rigid bodies	
`ent_rigid`	Rigid bodies	

After the function completes, you can easily iterate through the entity list to perform desired operations, as in the following code outline:

```
for (int i = 0; i < entityCount; \++i)
{

   IPhysicalEntity\* entity = entityList[i];

   [...]

   if (entity->GetType() == PE_RIGID)
   {
      [...]
   }

   [...]

}
```

If `ent_alloctate_list` is specified, the function allocates memory for the list (the memory can later be freed by a call to `pWorld->GetPhysUtils()->DeletePointer`). Otherwise, an internal pointer will be returned.

Note
Because the physics system uses this pointer in almost all operations that require forming an entity list, no such calls should be made when the list is in use. If such calls are required and memory allocation is undesired, copy the list to a local pre-allocated array before iterating over it.

Casting Rays in an Environment

The `RayWorldIntersection` physical world function casts rays into the environment.

Depending on the material that the ray hits and the ray properties, a hit can be pierceable or solid.

A pierceable hit is a hit that has a material pierceability higher than the ray's pierceability. Material pierceability and ray pierceability occupy the lowest 4 bits of `material` flags and `RayWorldIntersection` flags.

Pierceable hits don't stop the ray and are accumulated as a list sorted by hit distance. The caller provides the function with an array for the hits. A solid hit (if any) always takes the slot with index 0 and pierceable hits slots from 1 to the end.

Optionally, the function can separate between 'important' and 'unimportant' pierceable hits (importance is indicated by `sf_important` in material flags) and can make important hits have a higher priority (regardless of hit distance) than unimportant ones when competing for space in the array.

By default, `RayWorldIntersection` checks only entity parts with the `geom_colltype_ray` flag. You can specify another flag or combination of flags by setting `flags |= geom_colltype_mask<<rwi_colltype_bit`. In this case, all flags should be set in part so that the specified flag can be tested.

`RayTraceEntity` is a more low-level function and checks ray hits for one entity only. `RayTraceEntity` returns only the closest hit.

Alternatively, `CollideEntityWithBeam` can perform a sweep-check within a sphere of the specified radius. In order to detect collisions reliably, the sphere specified should be outside of the object. The `org` parameter corresponds to the sphere center.

Creating Explosions

The function `SimulateExplosion` is used to simulate explosions in a physical world.

The only effect of explosions inside the physics system are impulses that are added to the nearby objects. A single impulse is calculated by integrating impulsive pressure at an area fragment multiplied by this area and scaled by its orientation towards the epicenter.

Impulsive pressure has a falloff proportional to `1/distance2`. If `distance` is smaller than `rmin`, it is clamped to `rmin`.

`impulsive_pressure_at_r` is the impulsive pressure at distance `r`.

`SimulateExplosion` can optionally build an occlusion cubemap to find entities occluded from the explosion (`nOccRe_s` should be set to a non-zero cubemap resolution in one dimension in this case). First, static entities are drawn into the cubemap, and then dynamic entities of the types specified in `iTypes` are tested against the map. Thus, dynamic entities never occlude each other.

Passing -1 to `nOccRes` tells the function to reuse the cubemap from the last call and process only the dynamic entities that were not processed during the last call. This is useful when the code that creates the explosion decides to spawn new entities afterwards, such as debris or dead bodies, and wants to add explosion impulses to them without recomputing the occlusion map.

Due to the projective nature of the cubemap, small objects very close to the epicenter can occlude more than they normally would. To counter this, `rmin_occ` can specify linear dimensions of a small cube that is subtracted from the environment when building the occlusion map. This crops the smaller objects but can make explosions go through thin walls, so a compromise set of dimensions should be used.

`nGrow` specifies the number of occlusion cubemap cells (in one dimension) that dynamic entities are inflated with. This can help explosions to reach around corners in a controllable way. After a call has been made to `SimulateExplosion`, the physics system can return how much a particular entity was affected by by calling `IsAffectedByExplosion`.

`IsAffectedByExplosion` returns fraction from zero to one. The `IsAffectedByExplosion` function performs a lookup into a stored entity list; it does not recompute the cubemap. The explosion epicenter used for generating impulses can be made different from the one used to build a cubemap. For example, you can create explosions slightly below the ground to make things go up instead of sideways. Note that this function processes only parts with `geom_colltype_explosion`.

System

This section contains topics on general system issues, including memory handling, streaming, and localization. It also provides information on logging and console tools.

Topics

Memory Handling

This article discusses some memory and storage considerations related to game development.

Hardware Memory Limitations

Developing for game consoles can be challenging due to memory limitations. From a production point of view, it is tempting to use less powerful hardware for consoles, but the expectations for console quality are usually higher in an increasingly competitive market.

Choosing a Platform to Target

It is often better to choose only one development platform, even if multiple platforms are targeted for production. Choosing a platform with lower memory requirements eases production in the long run, but it can degrade the quality on other platforms. Some global code adjustments (for example, TIF setting "globalreduce", TIF preset setting "don't use highest LOD") can help in reducing memory usage, but often more asset-specific adjustments are needed, like using the TIF "reduce" setting. If those adjustments are insufficient, completely different assets are required (for example, all LODs of some object are different for console and PC). This can be done through a CryPak feature. It is possible to bind multiple pak files to a path and have them behave as layered. This way it is possible to customize some platforms to use different assets. Platforms that use multiple layers have more overhead (memory, performance, I/O), so it is better to use multiple layers on more powerful hardware.

Budgets

Budgets are mostly game specific because all kinds of memory (for example, video/system/disk) are shared across multiple assets, and each game utilizes memory differently. It's a wise decision to dedicate a certain amount of memory to similar types of assets. For example, if all weapons roughly cost the same amount of memory, the cost of a defined number of weapons is predictable, and with some careful planning in production, late and problematic cuts can be avoided.

Allocation Strategy with Multiple Modules and Threads

The Lumberyard memory manager tries to minimize fragmentation by grouping small allocations of similar size. This is done in order to save memory, allow fast allocations and deallocations and to minimize conflicts between multiple threads (synchronization primitives for each bucket). Bigger allocations run through the OS as that is quite efficient. It is possible to allocate memory in other than the main thread, but this can negatively impact the readability of the code. Memory allocated in one module should be deallocated in the same module. Violating this rule might work in some cases, but this breaks per module allocation statistics. The simple `Release()` method ensures objects are freed in the same module. The string class `(CryString)` has this behavior built in, which means the programmer doesn't need to decide where the memory should be released.

Caching Computational Data

In general, it is better to perform skinning (vertex transformation based on joints) of characters on the GPU. The GPU is generally faster in doing the required computations than the CPU. Caching the skinned result is still possible, but memory is often limited on graphics hardware, which tends to be stronger on computations. Under these conditions, it makes sense to recompute the data for every pass, eliminating the need to manage cache memory. This approach is advantageous because character counts can vary significantly in dynamic game scenes.

Compression

There are many lossy and lossless compression techniques that work efficiently for a certain kind of data. They differ in complexity, compression and decompression time and can be asymmetric. Compression can introduce more latency, and only few techniques can deal with broken data such as packet loss and bit-flips.

Disk Size

Installing modern games on a PC can be quite time consuming. Avoiding installation by running the game directly from a DVD is a tempting choice, but DVD performance is much worse than hard drive performance, especially for random access patterns. Consoles have restrictions on game startup times and often require a game to cope with a limited amount of disk memory, or no disk memory at all. If a game is too big to fit into memory, streaming is required.

Total Size

To keep the total size of a build small, the asset count and the asset quality should be reasonable. For production it can make sense to create all textures in double resolution and downsample the content with the Resource Compiler. This can be useful for cross-platform development and allows later release of the content with higher quality. It also eases the workflow for artists as they often create the assets in higher resolutions anyway. Having the content available at higher resolutions also enables the engine to render cut-scenes with the highest quality if needed (for example, when creating videos).

Many media have a format that maximizes space, but using the larger format can cost more than using a smaller one (for example, using another layer on a DVD). Redundancy might be a good solution to minimize seek times (for example, storing all assets of the same level in one block).

Address Space

Some operating systems (OSes) are still 32-bit, which means that an address in main memory has 32-bits, which results in 4 GB of addressable memory. Unfortunately, to allow relative addressing, the top bit is lost, which leaves only 2 GB for the application. Some OSes can be instructed to drop this limitation by compiling applications with large address awareness, which frees up more memory. However, the full 4 GB cannot be used because the OS also maps things like GPU memory into the memory space. When managing that memory, another challenge appears. Even if a total of 1 GB of memory is free, a contiguous block of 200 MB may not be available in the virtual address space. In order to avoid this problem, memory should be managed carefully. Good practices are:

- Prefer memory from the stack with constant size (SPU stack size is small).
- Allocating from the stack with dynamic size by using `alloca()` is possible (even on SPU), but it can introduce bugs that can be hard to find.
- Allocate small objects in bigger chunks (flyweight design pattern).
- Avoid reallocations (for example, reserve and stick to maximum budgets).
- Avoid allocations during the frame (sometimes simple parameter passing can cause allocations).
- Ensure that after processing one level the memory is not fragmented more than necessary (test case: loading multiple levels one after another).

A 64-bit address space is a good solution for the problem. This requires a 64-bit OS and running the 64-bit version of the application. Running a 32-bit application on a 64-bit OS helps very little. Note that compiling for 64-bit can result in a bigger executable file size, which can in some cases be counterproductive.

Bandwidth

To reduce memory bandwidth usage, make use of caches, use a local memory access pattern, keep the right data nearby, or use smaller data structures. Another option is to avoid memory accesses all together by recomputing on demand instead of storing data and reading it later.

Latency

Different types of memory have different access performance characteristics. Careful planning of data storage location can help to improve performance. For example, blending animation for run animation needs to be accessible within a fraction of a frame, and must be accessible in memory. In contrast, cut-scene animations can be stored on disk. To overcome higher latencies, extra coding may be required. In some cases the benefit may not be worth the effort.

Alignment

Some CPUs require proper alignment for data access (for example, reading a float requires an address dividable by 4). Other CPUs perform slower when data is not aligned properly (misaligned data access). As caches operate on increasing sizes, there are benefits to aligning data to the new sizes. When new features are created, these structure sizes must be taken into consideration. Otherwise, the feature might not perform well or not even work.

Virtual Memory

Most operating systems try to handle memory quite conservatively because they never know what memory requests will come next. Code or data that has not been used for a certain time can be paged out to the hard drive. In games, this paging can result in stalls that can occur randomly, so most consoles avoid swapping.

Streaming

Streaming enables a game to simulate a world that is larger than limited available memory would normally allow. A secondary (usually slower) storage medium is required, and the limited resource is used as a cache. This is possible because the set of assets tends to change slowly and only part of the content is required at any given time. The set of assets kept in memory must adhere to the limits of the hardware available. While memory usage can partly be determined by code, designer decisions regarding the placement, use, and reuse of assets, and the use of occlusion and streaming hints are also important in determining the amout of memory required. Latency of streaming can be an issue when large changes to the set of required assets are necessary. Seek times are faster on hard drives than on most other storage media like DVDs, Blue-Rays or CDs. Sorting assets and keeping redundant copies of assets can help to improve performance.

Split screen or general multi-camera support add further challenges for the streaming system. Tracking the required asset set becomes more difficult under these circumstances. Seek performance can get get worse as multiple sets now need to be supported by the same hardware. It is wise to limit gameplay so that the streaming system can perform well. A streaming system works best if it knows about the assets that will be needed beforehand. Game code that loads assets on demand without registering them first will not be capable of doing this. It is better to wrap all asset access with a handle and allow registration and creation of handles only during some startup phase. This makes it easier to create stripped down builds (minimal builds consisting only of required assets).

Streaming System

The Lumberyard streaming engine takes care of the streaming of meshes, textures, music, sounds, and animations.

Low-level Streaming System

CryCommon interfaces and structs

The file `IStreamEngine.h` in CryCommon contains all the important interfaces and structs used by the rest of the engine.

First of all there is the `IStreamEngine` itself. There is only one `IStreamingEngine` in the application and it controls all the possible I/O streams. Most of the following information comes directly from the documentation inside the code, so it's always good to read the actual code in `IStreamEngine.h` file for any missing information.

The most important function in `IStreamEngine` is the `StartRead` function which is used to start any streaming request.

IStreamEngine.h

```
UNIQUE_IFACE struct IStreamEngine
{
```

```
public:

    // Description:
    //   Starts asynchronous read from the specified file (the file may be on
a
    //   virtual file system, in pak or zip file or wherever).
    //   Reads the file contents into the given buffer, up to the given size.
    //   Upon success, calls success callback. If the file is truncated or for
 other
    //   reason can not be read, calls error callback. The callback can be NULL

    //   (in this case, the client should poll the returned IReadStream object;

    //   the returned object must be locked for that)
    // NOTE: the error/success/progress callbacks can also be called from INSIDE

    //   this function
    // Return Value:
    //   IReadStream is reference-counted and will be automatically deleted if

    //   you don't refer to it; if you don't store it immediately in an auto-
pointer,
    //   it may be deleted as soon as on the next line of code,
    //   because the read operation may complete immediately inside StartRead()

    //   and the object is self-disposed as soon as the callback is called.
    virtual IReadStreamPtr StartRead (const EStreamTaskType tSource, const char*
 szFile, IStreamCallback* pCallback = NULL, StreamReadParams* pParams = NULL)
= 0;

};
```

The following are the currently supported streaming task types. This enum should be extended if you want to stream a new object type.

IStreamEngine.h

```
enum EStreamTaskType
{
    eStreamTaskTypeCount      = 13,
    eStreamTaskTypePak        = 12, // Pak file itself
    eStreamTaskTypeFlash      = 11, // Flash file object
    eStreamTaskTypeVideo      = 10, // Video data (when streamed)
    eStreamTaskTypeReadAhead  = 9,  // Read ahead data used for file reading
prediction
    eStreamTaskTypeShader     = 8,  // Shader combination data
    eStreamTaskTypeSound      = 7,
    eStreamTaskTypeMusic      = 6,
    eStreamTaskTypeFSBCache   = 5,  // Complete FSB file
    eStreamTaskTypeAnimation  = 4,  // All the possible animations types (dba,
 caf, ..)
    eStreamTaskTypeTerrain    = 3,  // Partial terrain data
    eStreamTaskTypeGeometry   = 2,  // Mesh or mesh lods
    eStreamTaskTypeTexture    = 1,  // Texture mip maps (currently mip0 is not
 streamed)
};
```

A callback object can be provided to the `StartStream` function to be informed when the streaming request has finished. The callback object should implement the following `StreamAsyncOnComplete` and `StreamOnComplete` functions.

IStreamEngine.h

```
class IStreamCallback
{
public:
    // Description:
    //   Signals that reading the requested data has completed (with or without
 error).
    //   This callback is always called, whether an error occurs or not, and
is called
    //   from the async callback thread of the streaming engine, which happens

    //   directly after the reading operation
    virtual void StreamAsyncOnComplete (IReadStream* pStream, unsigned nError)
 {}

    // Description:
    //   Same as the StreamAsyncOnComplete, but this function is called from
the main
    //   thread and is always called after the StreamAsyncOnComplete function.

    virtual void StreamOnComplete (IReadStream* pStream, unsigned nError) = 0;
};
```

When starting a read request, you can also provide the optional parameters shown in the following code.

IStreamEngine.h

```
struct StreamReadParams
{
public:

    // The user data that'll be used to call the callback.
    DWORD_PTR dwUserData;

    // The priority of this read
    EStreamTaskPriority ePriority;

    // Description:
    //   The buffer into which to read the file or the file piece
    //   if this is NULL, the streaming engine will supply the buffer.
    // Notes:
    //   DO NOT USE THIS BUFFER during read operation! DO NOT READ from it, it
 can lead to memory corruption!
    void* pBuffer;

    // Description:
    //   Offset in the file to read; if this is not 0, then the file read
    //   occurs beginning with the specified offset in bytes.
    //   The callback interface receives the size of already read data as nSize

    //   and generally behaves as if the piece of file would be a file of its
own.
```

```
    unsigned nOffset;

    // Description:
    //   Number of bytes to read; if this is 0, then the whole file is read,
    //   if nSize == 0 && nOffset != 0, then the file from the offset to the
end is read.
    //   If nSize != 0, then the file piece from nOffset is read, at most nSize
 bytes
    //   (if less, an error is reported). So, from nOffset byte to nOffset +
nSize - 1 byte in the file.
    unsigned nSize;

    // Description:
    //   The combination of one or several flags from the stream engine general
 purpose flags.
    // See also:
    //   IStreamEngine::EFlags
    unsigned nFlags;
};
```

The return value of the `StartRead` function is an `IReadStream` object which can be optionally stored on the client. The `IReadStream` object is refcounted internally. When the callback object can be destroyed before the reading operation is finished, the readstream should be stored separately, and the abort should be called on it. Doing this will clean up the entire read request internally and will also call the async and sync callback functions.

The `Wait` function can be used to perform a blocking reading requests on the streaming engine. This function can be used from an async reading thread that uses the Lumberyard streaming system to perform the actual reading.

IStreamEngine.h

```
class IReadStream : public CMultiThreadRefCount
{
public:
    virtual void Abort() = 0;
    virtual void Wait( int nMaxWaitMillis=-1 ) = 0;
};
```

Internal flow of a read request

The Lumberyard stream engine uses extra worker and IO threads internally. For every possible IO input, a different `StreamingIOThread` is created which can run independently from the others.

Currently the stream engine has the following IO threads:

- Optical – Streaming from the optical data drive.
- Hard disk drive (HDD) – Streaming from installed data on the hard disk drive (this could be a fully installed game, or shadow copied data).
- Memory – Streaming from packed in-memory files, which requires very little IO.

When a reading request is made on the streaming engine, it first checks which IO thread to use, and computes the sortkey. The request is then inserted into one of the `StreamingIOThread` objects.

After the reading operation is finished, the request is forwarded to one of the decompression threads if the data was compressed, and then into one of the async callback threads. The amount of async callback threads is dependent on the platform, and some async callback threads are reserved for specific streaming request types such as geometry and textures. After the async callback has been processed, the finished streaming request is added to the streaming engine to be processed on the main thread. The next update on the streaming engine from the main thread will then call the sync callback (`StreamOnComplete`) and clean up the temporary allocated memory if needed.

For information regarding the IO/WorkerThreads please check the `StreamingIOThread` and `StreamingWorkerThread` class.

Read request sorting

Requests to the streaming engine are **not** processed in a the same order as which they have been requested. The system tries to internally 'optimize' the order in which to read the data, to maximize the read bandwidth.

When reading data from an optical disc , it is very important to reduce the amount of seeks. (This is also true when reading from a hard disk drive, but to a lesser extent). A single seek can take over 100 milliseconds, while the actual read time might take only a few milliseconds. Some official statistics from the 360 XDK follow.

- Outer diameter throughput : 12x (approximately 15 MB per second).
- Inner diameter throughput : 5x (6.8 MB per second).
- Average seek (1/3rd stroke) time : 110 ms typical, 140 ms maximum.
- Full stroke seek time : 180 ms typical, 240 ms maximum.
- Layer switch time : 75 ms.

The internal sorting algorithm takes the following rules into account in the following order.

- **Priority of the request** – High priority requests always take precedence, but too many of them can introduce too many extra seeks.
- **Time grouping** – Requests made within a certain time are grouped together to create a continuous reading operation on the disc for every time group. The default value is 2 seconds, but can be changed using the following cvar: `sys_streaming_requests_grouping_time_period`. Time grouping has a huge impact on the average completion time of the requests. It increases the time of a few otherwise quick reading requests, but drastically reduces the overall completion time because most of the streaming requests are coming from random places on the disc.
- **Actual offset on disc** – The actual disc offset is computed and used during the sorting. Files which have a higher offset get a higher priority, so it is important to organize the layout of the disc to reflect the desired streaming order.

For information regarding sorting, please refer to the source code in `StreamAsyncFileRequest::ComputeSortKey()`. The essential sorting code follows.

CAsyncIOFileRequest::ComputeSortKey

```
void CAsyncIOFileRequest::ComputeSortKey(uint64 nCurrentKeyInProgress)
{
    .. compute the disc offset (can be requested using CryPak)

   // group items by priority, then by snapped request time, then sort by disk
 offset
    m_nDiskOffset += m_nRequestedOffset;
    m_nTimeGroup = (uint64)(gEnv->pTimer->GetCurrTime() / max(1,
g_cvars.sys_streaming_requests_grouping_time_period));
    uint64 nPrioriry = m_ePriority;

    int64 nDiskOffsetKB = m_nDiskOffset >> 10; // KB
    m_nSortKey = (nDiskOffsetKB) | (((uint64)m_nTimeGroup) << 30) | (nPrioriry
 << 60);
}
```

Streaming statistics

The streaming engine can be polled for streaming statistics using the `GetStreamingStatistics()` function.

Most of the statistics are divided into two groups, one collected during the last second, and another from the last reset (which usually happens during level loading). Statistics can also be forcibly reset during the game.

The `SMediaTypeInfo` struct gives information per IO input system (hard disk drive, optical, memory).

IStreamEngine.h

```
struct SMediaTypeInfo
{
    // stats collected during the last second
    float fActiveDuringLastSecond;
    float fAverageActiveTime;
    uint32 nBytesRead;
    uint32 nRequestCount;
    uint64 nSeekOffsetLastSecond;
    uint32 nCurrentReadBandwidth;
    uint32 nActualReadBandwidth;    // only taking actual reading time into
account

    // stats collected since last reset
    uint64 nTotalBytesRead;
    uint32 nTotalRequestCount;
    uint64 nAverageSeekOffset;
    uint32 nSessionReadBandwidth;
    uint32 nAverageActualReadBandwidth; // only taking actual read time into
account
};
```

The `SRequestTypeInfo` struct gives information about each streaming request type, such as geometry, textures, and animations.

IStreamEngine.h

```
struct SRequestTypeInfo
{
    int nOpenRequestCount;
    int nPendingReadBytes;

    // stats collected during the last second
    uint32 nCurrentReadBandwidth;
    // stats collected since last reset
    uint32 nTotalStreamingRequestCount;
    uint64 nTotalReadBytes;     // compressed data
    uint64 nTotalRequestDataSize;   // uncompressed data
    uint32 nTotalRequestCount;
    uint32 nSessionReadBandwidth;

    float fAverageCompletionTime;   // Average time it takes to fully complete
 a request
    float fAverageRequestCount; // Average amount of requests made per second
};
```

The following example shows global statistics that contain all the gathered data.

IStreamEngine.h

```
struct SStatistics
{
    SMediaTypeInfo hddInfo;
    SMediaTypeInfo memoryInfo;
    SMediaTypeInfo opticalInfo;

    SRequestTypeInfo typeInfo[eStreamTaskTypeCount];

   uint32 nTotalSessionReadBandwidth;  // Average read bandwidth in total from
 reset - taking full time into account from reset
    uint32 nTotalCurrentReadBandwidth;  // Total bytes/sec over all types and
systems.

    int nPendingReadBytes;      // How many bytes still need to be read
    float fAverageCompletionTime;   // Time in seconds on average takes to
complete read request.
    float fAverageRequestCount; // Average requests per second being done to
streaming engine
    int nOpenRequestCount;      // Amount of open requests

    uint64 nTotalBytesRead;     // Read bytes total from reset.
   uint32 nTotalRequestCount;  // Number of request from reset to the streaming
 engine.

    uint32 nDecompressBandwidth;    // Bytes/second for last second

    int nMaxTempMemory;     // Maximum temporary memory used by the streaming
system
};
```

Streaming debug information

Different types of debug information can be requested using the following CVar: `sys_streaming_debug x`.

Streaming and Levelcache Pak Files

As mentioned earlier, it is very important to minimize the seeks and seek distances when reading from an optical media drive. For this reason, the build system is designed to optimize the internal data layout for streaming.

The easiest and fastest approach is to not do any IO at all, but read the data from compressed data in memory. For this, small paks for startup and each level are created. These are loaded into memory during level loading. Some paks remain in memory until the end of the level. Others are only used to speed up the level loading. All small files and small read requests should ideally be diverted to these paks.

A special `RC_Job` build file is used to generate these paks: `Bin32/rc/RCJob_PerLevelCache.xml`. These paks are generated during a normal build pipeline. The internal managment in the engine is done by the CResourceManager class, which uses the global `SystemEvents` to preload or unload the paks.

Currently, the following paks are loaded into memory during level loading (`sys_PakLoadCache`).

- **level.pak** – Contains all actual level data, and should not be touched after level loading anymore.
- **xml.pak**
- **dds0.pak** – Contains all lowest mips of all the textures in the level.
- **cgf.pak** and **cga.pak** – Only load when CGF streaming is enabled.

The following paks are cached into memory during the level load process (`sys_PakStreamCache`).

- **dds_cache.pak** - Contains all dds files smaller than 6 KB (except for dds.0 files).
- **cgf_cache.pak** - Contains all cgf files smaller than 32 KB (only when CGF streaming is enabled).

Important
Be sure that these paks are available. Without them, level loading can take up to a few minutes, and streaming performance is greatly reduced.

The information regarding all the resources of a level are stored in the `resourcelist.txt` and `auto_resourcelist.txt`. These files are generated by an automatic testing system which loads each level and executes a prerecorded playthrough on it. These `resourcelist` files are used during the build phase to generate the level paks.

All data not in these in memory paks is handled through IO on the optical drive or hard disk drive, and it is also best to reduce the amount of seeks here. This optimization phase is also performed during the build process using the resource compiler.

All the data which can be streamed is extracted from all the resource lists from all levels, and is removed from the default pak files (for example, `objects.pak`, `textures.pak`, `animations.pak`) and put into new optimized paks for streaming inside a streaming folder.

The creating of the streaming paks uses the following rules:

- Split by **extension**: Different extension files are put into different paks (for example, dds, caf, dba, cgf) so that files of the same type can be put close to each other. This enables them to be read in bursts. The paks are also used to increase the priority of certain file types during request sorting by using the disc offset.
- Split by DDS **type**: Different dds types are sorted differently to increase the priority of different types (for example, diffuse maps get higher priority than normal maps). The actual distance in the pak is used during the sorting of the request.
- Split by DDS **mip**: The highest mips are put into a separate pak file. They usually take more than 60% of the size of all the smaller mips and can then be streamed with a lower priority. This greatly reduces the average seek time required to read the smaller textures. The texture streaming system internally optimizes the reads to reflect these split texture data.
- Sort **alphabetically**: Default alphabetical sorting is required because some of the data (such as CGF's during MP level loading), are loaded in alphabetical order. Changing this sort order can have a severe impact on the loading times.

The actual sorting code is hardcoded in the resource compiler, and can be found at: `Code\Tools\RC\ResourceCompiler\PakHelpers.cpp`.

Important
If you make changes to the sorting operator in the resource compiler, be sure to make the same changes to the texture streaming and streaming engine sorting operators.

Single Thread IO Access and Invalid File Access

It is very important that only a single thread access a particular IO device at one time. If multiple threads read from the same IO device concurrently, then the reading speed is more than halved, and it may take a number of seconds to read just a few kilobytes. This occurs because the IO reading head will partially read a few kilobytes for one thread, and then read another few kilobytes for another thread while always performing expensive seeks in between.

The solution is to exclusively read from `StreamingIOThreads` during gameplay. Lumberyard will by default show an **Invalid File Access** warning in the top left corner when reading data from the wrong thread, and will stall deliberately for threed seconds to emulate the actual stall when reading from an optical drive.

High Level Streaming Engine Usage

It is very easy to extend the current streaming functionality using the streaming engine. In this section, a small example class is presented that shows how to add a new streaming type.

First, create a class which derives from the `IStreamCallback` interface, which informs about streaming completion, and add some basic functionality to read a file. The file can either be read directly or use the streaming engine. When the data is read directly, it calls the `ProcessData` function to parse the loaded data. The function is also called from the `async` callback. Some processing can be performed here on the data if needed because it does not run on the main thread.

The default parameters are used when starting a reading request on the streaming engine. It is also possible to specify the final data storage to help reduce the number of dynamic allocations performed by the streaming engine.

The class also stores the read stream object in order to get information about the streaming request or to be able to cancel the request when the callback object is destroyed. The pointer is reset in the sync callback because after the call it will no longer be referenced by the streaming engine.

CNewStreamingType

```
#include
class CNewStreamingType : public IStreamCallback
{
public:
    CNewStreamingType() : m_pReadStream(0), m_bIsLoaded(false) {}
    ~CNewStreamingType()
    {
        if (m_pReadStream)
            m_pReadStream->Abort();
    }

    // Start reading some data
    bool ReadFile(const char* acFilename, bool bUseStreamingEngine)
    {
        if (bUseStreamingEngine)
        {
            StreamReadParams params;
            params.dwUserData = eLoadFullData;
            params.ePriority = estpNormal;
            params.nSize = 0; // read the full file
            params.pBuffer = NULL; // don't provide any buffer, but copy data
when streaming is done

          m_pReadStream = g_pISystem->GetStreamEngine()->StartRead(eStreamTask
TypeNewType, acFilename, this, &params);
        }
        else
        {
            // old way of reading file in a sync way (blocking call!)
            const char* acData = 0;
            size_t stSize = 0;

            .. read file directly using CryPak or fopen/fread

            ProcessData(acData, stSize);
            m_bIsLoaded = true;
        }

        return m_bIsLoaded;
    }

    // Check if the data is ready and loaded
    bool IsLoaded() const { return m_bIsLoaded; }

protected:

    // implement the IStreamCallback function
    void StreamAsyncOnComplete(IReadStream* pStream, unsigned nError)
    {
        if(nError)
        {
            return;
        }

        const char* acData = (char*)pStream->GetBuffer();
        size_t stSize= pStream->GetBytesRead();

        ProcessData(acData, stSize);
        m_bIsLoaded = true;
    }

    void StreamOnComplete (IReadStream* pStream, unsigned nError)
    {
        m_pReadStream = 0;
    }

    // process the actual loaded data
    void ProcessData(const char* acData, size_t stSize);

    // store the stream callback object to be sure it can be canceled when the
 object is destroyed
    IReadStreamPtr m_pReadStream;
    // Extra flag used to check if the data is ready
    bool m_bIsLoaded;
}
```

Text Localization and Unicode Support

Because games are typically localized to various languages, your game might have to use text data for many languages.

This document provides programming-related information regarding localization, including localization information specific to Lumberyard.

Terminology

The following table provides brief descriptions of some important terms related to localization and text processing.

Term	Description
character	A unit of textual data. A character can be a glyph or formatting indicator. Note that a glyph does not necessarily form a single visible unit. For example, a diacritical mark [´] and the letter [a] are separate glyphs (and characters), but can be overlaid to form the character [á].
Unicode	A standard maintained by the Unicode Consortium that deals with text and language standardization.
UCS	Universal Character Set, the standardized set of characters in the Unicode standard (also, ISO-10646)
(UCS) code-point	An integral identifier for a single character in the UCS defined range, typically displayed with the U prefix followed by hexadecimal, for example: U+12AB
(text) encoding	A method of mapping (a subset of) UCS to a sequence of code-units, or the process of applying an encoding.
code-unit	An encoding-specific unit integral identifier used to encode code-points. Many code-units may be used to represent a single code-point.
ASCII	A standardized encoding that covers the first 128 code-points of the UCS space using 7- or 8-bit code-units.
(ANSI) code-page	A standardized encoding that extends ASCII by assigning additional meaning to the higher 128 values when using 8-bit code-units There are many hundreds of code-pages, some of which use multi-byte sequences to encode code-points.
UTF	UCS Transformation Format, a standardized encoding that covers the entire UCS space.
UTF-8	A specific instance of UTF, using 8-bit code-units. Each code-point can take 1 to 4 (inclusive) code-units.
UTF-16	A specific instance of UTF, using 16-bit code-units. Each code-point can take 1 or 2 code-units.
UTF-32	A specific instance of UTF, using 32-bit code-units. Each code-point is directly mapped to a single code-unit.

Term	Description
byte-order	How a CPU treats a sequence of bytes when interpreting multi-byte values. A byte-orderTypically either little-endian or big-endian format
encoding error	A sequence of code-units that does not form a code-point (or an invalid code-point, as defined by the Unicode standard)

What encoding to use?

Since there are many methods of encoding text, the question that should be asked when dealing with even the smallest amount of text is, "In what encoding is this stored?" This is an important question because decoding a sequence of code-units in the wrong way will lead to encoding errors, or even worse, to valid decoding that yields the wrong content.

The following table describes some common encodings.

Encoding	Code-unit size	Code-point size	Maps the entire UCS space	Trivial to encode/decode	Immune to byte-order differences	Major users
ASCII	7 bits	1 byte	no	yes	yes	Many English-only apps
(ANSI) code-page	8 bits	varies, usually 1 byte	no	varies, usually yes	yes	Older OS functions
UTF-8	8 bits	1 to 4 bytes	yes	no	yes	Most text on the internet, XML
UTF-16	16 bits	2 to 4 bytes	yes	yes	no	Windows "wide" API, Qt
UCS-2	16 bits	2 bytes	no	yes	no	None (replaced with UTF-16)
UTF-32 UCS-4	32 bits	4 bytes	yes	yes	no	Linux "wide" API

Because there is no single "best" encoding, you should always consider the scenario in which it will be used when choosing one.

Historically, different operating systems and software packages have chosen different sets of supported encodings. Even C++ follows different conventions on different platforms. For example, the "wide character" `wchar_t` is 16-bits on Windows, but 32-bits on Linux.

Because Lumberyard products can be used on many platforms and in many languages, full UCS coverage is desirable. The follow table presents some conventions used in Lumberyard:

Text data type	Encoding	Reason
Source code	ASCII	We write our code in English, which means ASCII is sufficient.

Text data type	Encod-ing	Reason
Text assets	UTF-8	Assets can be transferred between machines with potentially differing byte-order, and may contain text in many languages.
Run-time variables	UTF-8	Since transforming text data from or to UTF-8 is not free, we keep data in UTF-8 as much as possible. Exceptions must be made when interacting with libraries or operating systems that require another encoding. In these cases all transformations should be done at the call-site.
File and path names	ASCII	File names are a special case with regards to case-sensitivity, as defined by the file system. Unicode defines 3 cases, and conversions between them are locale-specific. In addition, the normalization formats are typically not (all) accounted for in file-systems and their APIs. Some specialized file-systems only accept ASCII. This combination means that using the most basic and portable sub-set should be preferred, with UTF-8 being used only as required.

General principles

- Avoid using non-ASCII characters in source code. Consider using escape sequences if a non-ASCII literal is required.
- Avoid using absolute paths. Only paths that are under developer control should be entered. If possible, use relative ASCII paths for the game folder, root folder, and user folder. When this is not possible, carefully consider non-ASCII contents that may be under a user's control, such as those in the installation folder.

How does this affect me when writing code?

Since single-byte code-units are common (even in languages that also use double-byte code-units), single-byte string types can be used almost universally. In addition, since Lumberyard does not use ANSI code-pages, all text must be either ASCII or UTF-8.

The following properties hold for both ASCII and UTF-8.

- The NULL-byte (integral value 0) only occurs when a NULL-byte is intended (UTF-8 never generates a NULL-byte as part of multi-byte sequences). This means that C-style null-terminated strings act the same, and CRT functions like `strlen` will work as expected, except that it counts code-units, not characters.
- Code-points in the ASCII range have the same encoded value in UTF-8. This means that you can type English string literals in code and treat them as UTF-8 without conversion. Also, you can compare characters in the ASCII range directly against UTF-8 content (that is, when looking for an English or ASCII symbol sub-string).
- UTF-8 sequences (containing zero or more entire code-points) do not carry context. This means they are safe to append to each other without changing the contents of the text.

The difference between position and length in code-units (as reported through `string::length()`, `strlen()`, and similar functions) and their matching position and length in code-points is largely irrelevant. This is because the meaning of the sequence is typically abstract, and the meaning of the bytes matters only when the text is interpreted or displayed. However, keep in mind the following caveats.

- **Splitting strings** – When splitting a string, it's important to do one of the following.
  1. Recombine the parts in the same order after splitting, without interpreting the splitted parts as text (that is, without chunking for transmission).

2. Perform the split at a boundary between code-points. The positions just before and just after any ASCII character are always safe.

- **API boundaries** – When an API accepts or returns strings, it's s important to know what encoding the API uses. If the API doesn't treat strings as opaque (that is, interprets the text), passing UTF-8 may be problematic for APIs that accept byte-strings and interpret them as ASCII or ANSI. If no UTF-8 API is available, prefer any other Unicode API instead (UTF-16 or UTF-32). As a last resort, convert to ASCII, but understand that the conversion is lossy and cannot be recovered from the converted string. Always read the documentation of the API to see what text encoding it expects and perform any required conversion. All UTF encodings can be losslessly converted in both directions, so finding any API that accepts a UTF format gives you a way to use UTF encoding.
- **Identifiers** – When using strings as a "key" in a collection or for comparison, avoid using non-ASCII sequences as keys, as the concept of "equality" of UTF is complex due to normalization forms and locale-dependent rules. However, comparing UTF-8 strings byte-by-byte is safe if you only care about equality in terms of code-points (since code-point to code-unit mapping is 1:1).
- **Sorting** – When using strings for sorting, keep in mind that locale-specific rules for the order of text are complex. It's fine to let the UI deal with this in many cases. In general, make no assumptions of how a set of strings will be sorted. However, sorting UTF-8 strings as if they were ASCII will actually sort them by code-point. This is fine if you only require an arbitrary fixed order for `std::map` look-up, but displaying contents in the UI in this order may be confusing for end-users that expect another ordering.

In general, avoid interpreting text if at all possible. Otherwise, try to operate on the ASCII subset and treat all other text parts as opaque indivisible sequences. When dealing with the concept of "length" or "size", try to consider using in code-units instead of code-points, since those operations are computationally cheaper. In fact, the concept of the "length" of Unicode sequences is complex, and there is a many-to-many mapping between code-points and what is actually displayed.

How does this affect me when dealing with text assets?

In general, always:

- Store text assets with UTF-8 encoding.
- Store with Unicode NFC (Normalization Form C). This is the most common form of storage in text editing tools, so it's best to use this form unless you have a good reason to do otherwise.
- Store text in the correct case (that is, the one that will be displayed). Case-conversion is a complex topic in many languages and is best avoided.

Utilities provided in CryCommon

Lumberyard provides some utilities to make it easy to losslessly and safely convert text between Unicode encodings. In-depth technical details are provided in the header files that expose the `UnicodeFunctions.h` and `UnicodeIterator.h` utilities.

The most common use cases are as follows.

```
string utf8;
wstring wide;
Unicode::Convert(utf8, wide); // Convert contents of wide string and store into
 UTF-8 string
Unicode::Convert(wide, utf8); // Convert contents of UTF-8 string to wide string
```

```
string ascii;
Unicode::Convert<Unicode::eEncoding_ASCII, Unicode::eEncoding_UTF8>(ascii,
utf8); // Convert UTF-8 to ASCII (lossy!)
```

Important
The above functions assume that the input text is already validly encoded. To guard against malformed user input or potentially broken input, consider using the `Unicode::ConvertSafe` function.

Further reading

For an introduction to Unicode, see The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!).

For official information about Unicode, see The Unicode Consortium.

CryLog

CryLog Logging Functionality

You can log in Lumberyard by using the following global functions.

- `CryLog (eMessage)`
- `CryLogAlways (eAlways)`
- `CryError (eError)`
- `CryWarning (eWarning)`
- `CryComment (eComment)`

If more control is required, the `ILog` interface can be used directly by using the following syntax.

```
gEnv->pLog->LogToFile("value %d",iVal);
```

Verbosity Level and Coloring

You can control the verbosity of logging with the console variables `log_Verbosity` and `log_FileVerbosity`.

The following table shows the levels of verbosity and color convention. In the console, warnings appear in yellow, and errors appear in red.

Message	verbosity 0	verbosity 1	[illegible]
eAlways	X	X	X
eErrorAlways	X	X	de X

Message	verbosity 0	verbosity 1	-breC r so yt i -mo3 elos
eWarningAlways	X	X	- leX wo l
eInput	?	?	?
eInputResponse	?	?	?
eError		X	de X
eWarning	-	-	- leX wo l
eMessage	-	-	X
eComment	-	-	X

Key

- X – the message type is logged to the console or file
- ? – some special logic is involved

Tip
Full logging (to console and file) can be enabled by using `log_Verbosity 4`.

Log Files

The following log file sources write to the log files indicated.

Source	Log file
Lumberyard Editor	`Editor.log`
Game	`game.log` (default)
Error messages	`Error.log`

Console Variables

The following console variables relate to logging.

`log_IncludeTime`

```
Toggles time stamping of log entries.
Usage: log_IncludeTime [0/1/2/3/4/5]
  0=off (default)
  1=current time
  2=relative time
  3=current+relative time
  4=absolute time in seconds since this mode was started
  5=current time+server time
```

ai_LogFileVerbosity

```
None = 0, progress/errors/warnings = 1, event = 2, comment = 3
```

log_Verbosity DUMPTODISK

```
defines the verbosity level for log messages written to console
-1=suppress all logs (including eAlways)
0=suppress all logs(except eAlways)
1=additional errors
2=additional warnings
3=additional messages
4=additional comments
```

CryConsole

The console is a user interface system which handles console commands and console variables. It also outputs log messages and stores the input and output history.

Color coding

The game console supports color coding by using the color indices 0..9 with a leading $ character. The code is hidden in the text outputted on the console. Simple log messages through the `ILog` interface can be used to send text to the console.

```
This is normal $1one$2two$3three and so on
```

In the preceeding example, one renders in red, two in green, and three (and the remaining text) in blue.

Dumping all console commands and variables

All console commands and console variables can be logged to a file by using the command `DumpCommandsVars`. The default filename is `consolecommandsandvars.txt`.

To restrict the variables that should be dumped, a sub-string parameter can be passed. For example, the command

```
DumpCommandsVars i_
```

logs all commands and variables that begin with the sub-string "i_". (for example, `i_giveallitems` and `i_debug_projectiles`).

Console Variables

Console variables provide a convenient way to expose variables which can be modified easily by the user either by being entered in the console during runtime or by passing it as command-line argument before launching the application.

More information on how to use command-line arguments can be found in the Command Line Arguments article.

Console variables are commonly referred to as `CVar` in the code base.

Registering new console variables

For an integer or float based console variable, it is recommended to use the `IConsole::Register()` function to expose a C++ variable as a console variable.

To declare a new string console variable, use the `IConsole::RegisterString()` function.

Accessing console variables from C++

Console variables are exposed using the `ICVar` interface. To retrieve this interface, use the `IConsole::GetCVar()` function.

The most efficient way to read the console variable value is to access directly the C++ variable bound to the console variable proxy.

Adding New Console Commands

The console can easily be extended with new console commands. A new console command can be implemented in C++ as a static function which follows the `ConsoleCommandFunc` type. Arguments for this console command are passed using the `IConsoleCmdArgs` interface.

The following code shows the skeleton implementation of a console command:

```
static void RequestLoadMod(IConsoleCmdArgs* pCmdArgs);

void RequestLoadMod(IConsoleCmdArgs* pCmdArgs)
{
  if (pCmdArgs->GetArgCount() == 2)
  {
  const char* pName = pCmdArgs->GetArg(1);
  // ...
  }
  else
  {
  CryLog("Error, correct syntax is: g_loadMod modname");
  }
}
```

The following code will register the command with the console system:

```
IConsole* pConsole = gEnv->pSystem->GetIConsole();
pConsole->AddCommand("g_loadMod", RequestLoadMod);
```

Console Variable Groups

Console variable groups provide a convenient way to apply predefined settings to multiple console variables at once.

Console variables are commonly referred to as `CVarGroup` in the code base. Console variable groups can modify other console variables to build bigger hierarchies.

Warning
Cycles in the assignments are not detected and can cause crashes.

Registering a new variable group

To register a new variable group, add a new `.cfg` text file to the `GameSDK\config\CVarGroups` directory.

`sys_spec_Particles.cfg`

```
[default]
; default of this CVarGroup
= 4
e_particles_lod=1
e_particles_max_emitter_draw_screen=64

[1]
e_particles_lod=0.75
e_particles_max_emitter_draw_screen=1

[2]
e_particles_max_emitter_draw_screen=4

[3]
e_particles_max_emitter_draw_screen=16
```

This creates a new console variable group named `sys_spec_Particles` that behaves like an integer console variable. By default, this variable has the state `4` (set in the line following the comment in the example).

On changing the variable, the new state is applied. Console variables not specified in the `.cfg` file are not set. All console variables need to be part of the default section. An error message is output in case of violation of this rule.

If a console variable is not specified in a custom section, the value specified in the default section is applied.

Console variable group documentation

The documentation of the console variable group is generated automatically.

`sys_spec_Particles`

```
Console variable group to apply settings to multiple variables

sys_spec_Particles [1/2/3/4/x]:
 ... e_particles_lod = 0.75/1/1/1/1
 ... e_particles_max_screen_fill = 16/32/64/128/128
 ... e_particles_object_collisions = 0/1/1/1/1
 ... e_particles_quality = 1/2/3/4/4
 ... e_water_ocean_soft_particles = 0/1/1/1/1
 ... r_UseSoftParticles = 0/1/1/1/1
```

Checking if a console variable group value represents the state of the variables it controls

From the console

In the console you can type in the console variable group name and press tab. If the variable value is not represented, it will print the value of `RealState`.

```
sys_spec_Particles=2 [REQUIRE_NET_SYNC] RealState=3
sys_spec_Sound=1 [REQUIRE_NET_SYNC] RealState=CUSTOM
sys_spec_Texture=1 [REQUIRE_NET_SYNC]
```

By calling the console command `sys_RestoreSpec` you can check why the `sys_spec_` variables don't represent the right states.

From C++ code

From the code you can use the member function `GetRealIVal()` and compare its return value against the result of `GetIVal()` in `ICVar`.

Deferred execution of command line console commands

The commands that are passed via the command line by using the + prefix are stored in a separate list as opposed to the rest of the console commands.

This list allows the application to distribute the execution of those commands over several frames rather than executing everything at once.

Example

Consider the following example.

```
--- autotest.cfg --
hud_startPaused = "0"
wait_frames 100
screenshot autotestFrames
wait_seconds 5.0
screenshot autotestTime

-- console --
crysis.exe -devmode +map island +exec autotest +quit
```

In the example, the following operations were performed:

- Load the island map.
- Wait for 100 frames.
- Take a screenshot called autotestFrames.

- Wait for 5 seconds.
- Take a screenshot called autotestTime.
- Quit the application.

Details

Two categories of commands are defined: blocker and normal.

For each frame, the deferred command list is processed as a fifo. Elements of this list are consumed as long as normal commands are encountered.

When a blocker is consumed from the list and executed, the process is delayed until the next frame. For instance, commands like `map` and `screenshot` are blockers.

A console command (either command or variable) can be tagged as a blocker during its declaration using the `VF_BLOCKFRAME` flag.

The following synchronization commands are supported.

Optional Title

Command	Type	Description
`wait_frames` *`num`*`:`	<int>	Wait for *`num`* frames before the exeution of the list is resumed.
`wait_seconds` *`sec`*`:`	<float>	Wait for *`sec`* seconds before the exeution of the list is resumed.

CVar Tutorial

This tutorial shows you how to modify existing and create console variables (CVars). CVars can be used to control many configurable behaviors in Lumberyard. You can also use them in your game.

Note
This brief tutorial is intended for programmers. Most of the content uses code.

Creating CVars

To create a console variable

1. In your code editor, open the `Code\GameSDK\GameDll\GameCVars.h` file, which declares all game-specific CVars.
2. Locate the `SCVars` struct. Inside the struct, declare a new variable, as in the following example.

```
struct SCVars
{
 int g_tutorialVar; //add this line

 //... pre-existing code ...
};
```

The variable you added will be used to store the current value of the variable. If you need to store fractional numbers, you can also add a variable of the type `float`.

Next, you will register the CVar with the game engine so that its value can be changed by using the console.

3. In the same `Code\GameSDK\GameDll\GameCVars.cpp` file, locate the `InitCVars` function.

```
void SCVars::InitCVars(IConsole *pConsole)
{
 m_releaseConstants.Init( pConsole );

 REGISTER_CVAR(g_tutorialVar, 42, VF_NULL, "This CVar was added using the
tutorial on CVars"); //add this line

    //... pre-existing code ...
}
```

4. Specify a default value and help text for the variable. You can initialize the variable with any value that is valid for the type with which the variable was declared in the header file. The preceeding example specifies 42 as the default value and some help text that will be shown to users.
5. When your game unloads, be sure to un-register the variable. In the `Code\GameSDK\GameDll\GameCVars.cpp` file, locate and use the `ReleaseCVars` function, as shown in the following example.

```
void SCVars::ReleaseCVars()
{
 IConsole* pConsole = gEnv->pConsole;

 pConsole->UnregisterVariable("g_tutorialVar", true); //add this line

 //... pre-existing code ...
}
```

6. After you finish making changes, don't forget to compile your code.

Using the CVar

You can now change the value of the CVar that you created by using code, the console, and `.cfg` files.

From code

To access the value of the variable in your game code, use the `g_pGameCVars` pointer, as shown in the following example.

```
int myTutorialVar = g_pGameCVars->g_tutorialVar;
```

From the console

To change the value of the cvar from the console, use the syntax *`cvar_name`*=*`cvar_value`*. The following example sets the value of the `g_tutorialVar` console variable to 1337.

```
g_tutorialVar = 1337
```

From .cfg files

It's also possible to change the default CVar value from one of the `.cfg` files. Whenever a CVar is assigned a value, its previous value is discarded. Therefore, the last assignment is the one that is current.

The following list shows the order of initialization for console variables.

1. The value specified in the `GameCVars.cpp` file when `REGISTER_CVAR` is used. (A change here requires compiling.)
2. The value specified in the `system.cfg` file.
3. The value specified in the user's `user.cfg` file.
4. Any value assigned at game runtime.

Tip
To change the default value of an existing CVar without having to compile, add a line to `system.cfg` file to override the default.