**Scenario File**

**Format**: INI File.   
**Extension**: \*.scen

Scenario files link scripts together into a playable scenario.  
Each scenario file must be placed in the /Data/Scenarios directory for the game to recognize it.

**[General]**

|  |  |  |  |
| --- | --- | --- | --- |
| Key | Type | Default Value | Description |
| Name | **string** |  | The name of the scenario. This will be displayed in the mission selection screen. |
| PlayerName | **string** | “Luke” | The name other actors will call the player craft. |
| Description | **string** |  | The description of the scenario. This will be displayed in the mission selection screen. |
| Wings | **string[]** |  | The list of craft IDs the player is allowed to choose for this mission. |
| Difficulties | **string[]** |  | The list of difficulties the player is allowed to choose for this mission. The difficulty adjustment must be implemented by the scenario itself. |

**[Bindings]**

|  |  |  |  |
| --- | --- | --- | --- |
| Key | Type | Default Value | Description |
| Fn\_load | **string** | “load” | The name of the load function. This function is called at initialisation, after Fn\_loadfaction and Fn\_loadscene. |
| Fn\_loadfaction | **string** | “loadfaction” | The name of the loadfaction function. This function is called at initialisation. |
| Fn\_loadscene | **string** | “loadscene” | The name of the loadscene function. This function is called at initialisation, after Fn\_loadfaction. |
| Fn\_makeplayer | **string** | “makeplayer” | The name of the makeplayer function. Defaults to 'makeplayer'. This function is called every time a player is supposed to respawn, both initially and after each death. The actual respawn logic must be implemented by the scenario itself. |
| Fns\_gametick | **string[]** |  | The list of the tick functions. This function is called every tick of the game. |

**[Bindings]**

|  |  |  |  |
| --- | --- | --- | --- |
| Key | Type | Default Value | Description |
| Win | **string** |  | The audio played with the scenario is won. This path is relative to the /assets/music folder, and excludes the \*.mp3 extension. |
| Lose | **string** |  | The audio played with the scenario is lost. This path is relative to the /assets/music folder, and excludes the \*.mp3 extension. |

**[Scripts]**

The entries in Scripts are a list of script files to be included in this scenario. Each new line corresponds to a new file.  
Each entry is a relative path from the executable's location.  
Each file must be in a script file format.

|  |
| --- |
| Example |
| [Scripts]  Common\Spawn.sw  Common\ActorHealth.sw  Imperial\13\_8\Script.sw  Imperial\13\_8\Message.sw |

**Script File**

**Format**: Script File.   
**Extension**: typically \*.sw

Script files are used by the game to define programmable logic.  
They have a custom syntax that is interpreted by the game engine to generate behaviour. This way you can define your own scenario without needing to modify the game program.

**Syntax**

Script files take a pseudo-C style language with similar implementations.

A script may look a bit like this:

|  |
| --- |
| Example |
| float3 faction\_empire\_color = { 0, 0.8, 0 };  int tiea1;  loadfaction:  Faction.Add("Empire", faction\_empire\_color);  load:  // spawns the player  string tiea1\_type = GetPlayerActorType();  float3 tiea1\_pos = {100, 0, 10000};  float3 tiea1\_rot = {0, -180, 0};  tiea1 = Actor.Spawn(tiea1\_type, "Alpha-1", "", "", 0, "Empire", tiea1\_pos, tiea1\_rot);  Actor.SetProperty(tiea1, " Health.MaxShd", 25);  Actor.QueueLast(tiea1, "wait", 2.5);  Actor.AddToSquad(Player.GetActor(), tiea1); |

The base structure of a script takes the form of a series of statements separated by script headings:

|  |
| --- |
| **// global statements**  **float3** faction\_empire\_color = { 0, 0.8, 0 };  **int** tiea1; |
| **// script header - loadfaction**  **loadfaction:**  **// statements within script “loadfaction”**  **Faction.Add**("Empire", faction\_empire\_color); |
| **// script header - load**  **load:**  **// statements within script “load”**  // spawns the player  **string** tiea1\_type = **GetPlayerActorType**();  **float3** tiea1\_pos = {100, 0, 10000};  **float3** tiea1\_rot = {0, -180, 0};  tiea1 = **Actor.Spawn**(tiea1\_type, "Alpha-1", "", "", 0, "Empire", tiea1\_pos, tiea1\_rot);  **Actor.SetProperty**(tiea1, " Health.MaxShd", 25);  **Actor.QueueLast**(tiea1, "wait", 2.5);  **Actor.AddToSquad**(**Player.GetActor**(), tiea1); |

The script file begins with global statements, until a script header is encountered.

|  |
| --- |
| **// global statements**  **float3** faction\_empire\_color = { 0, 0.8, 0 };  **int** tiea1; |

The script header contains the name of the script, followed by the colon (':') sign.

|  |
| --- |
| **loadfaction:** |

The script header is followed by script body which consists of a number of statements, until either another header or end of file is encountered.

|  |
| --- |
| **loadfaction:**  **Faction.Add**("Empire", faction\_empire\_color); |

Each statement is terminated by the semicolon (';') literal.

|  |
| --- |
| **string** tiea1\_type = **GetPlayerActorType**();  **float3** tiea1\_pos = {100, 0, 10000};  **float3** tiea1\_rot = {0, -180, 0}; |

A statement may span multiple times, but must be ended by the semicolon (';') literal.

|  |
| --- |
| tiea1 = **Actor.Spawn**(  tiea1\_type,  "Alpha-1",  "",  "",  0,  "Empire",  tiea1\_pos,  tiea1\_rot); |

A statement usually comprises one or more expression phrases, which is a basic block of operations.  
An expression usually comprises one or more literal instances, and some operator or function block.

|  |
| --- |
| Examples of statements |
| float3 faction\_empire\_color = { 0, 0.8, 0 };  loadfaction:    load:  // spawns the player  string tiea1\_type = GetPlayerActorType();  float3 tiea1\_pos = {100, 0, 10000};  float3 tiea1\_rot = {0, -180, 0};  tiea1 = Actor.Spawn(tiea1\_type, "Alpha-1", "", "", 0, "Empire", tiea1\_pos, tiea1\_rot);  Actor.SetProperty(tiea1, " Health.MaxShd", 25);  Actor.QueueLast(tiea1, "wait", 2.5);  Actor.AddToSquad(Player.GetActor(), tiea1); |
| bool enabled; |
| Faction.Add("Empire", faction\_empire\_color); |
| float[] actor\_ids = {12, 24, 360}; |
| j += 1; |

|  |
| --- |
| Examples of expressions |
| { 0, 1, 2 } |
| 2 + 1 |
| "Alpha-” + i |
| actor\_hp > 10 |
|  |

A statement follows C-style lexicon. The following are possible:

[LITERAL] Keywords:

if Used in a if-else-then syntatic structure

else Used in a if-else-then syntatic structure

then Used in a if-else-then syntatic structure

foreach Used in a foreach-in syntatic structure

in Used in a foreach-in syntatic structure

[LITERAL] Use of the following literals:

bool Represented by (true|false)

int Represented by both decimal format (10) and hexadecimal format (0x0100)

float Represented by a decimal format (0.025)

string Represented by quoted strings ("example")

variable Represented by any unquoted string beginning with a character beginning with alphabet (A-Z, a-Z) or

an underscore (\_). Keywords are reserved and cannot be used as variable names.

Variables are weak-typed. You may assign an int value to a variable that was previously assigned to

a bool value. However, take care to feed the correct types to your operators and functions to avoid

runtime errors!

[EXPRESSION] Unary operator expressions:

+x Identity

-x Negation. Use on numeric operands only.

!x Logical Negation. Use on bool operands only.

~x Alias of [!x]

[EXPRESSION] Binary operator expressions:

x + y Returns the sum of x and y if both are numeric, or the concatenation of x and y if at least one of

them is a string. Does not work on bool or other formats.

x - y Returns the difference of x and y if both are numeric, the result is negative if y is larger than x.

Does not work on bool, string or other formats.

x \* y Returns the multiplicative result of x and y if both are numeric. Does not work on bool, string or

other formats.

x / y Returns the division result of x by y if both are numeric. Does not work on bool, string or other

formats.

x % y Returns the modulus result of x and y if both are numeric. Does not work on bool, string or other

formats.

x || y Returns the logical OR of x and y if both are bool. This operation performs lazy evaluation; y need

not be evaluated if x is true. Does not work on numeric, string or other formats.

x && y Returns the logical AND of x and y if both are bool. This operation performs lazy evaluation; y need

not be evaluated if x is false. Does not work on numeric, string or other formats.

x == y Returns true if x and y are equal. Incompatible types return false (float and int are compatible for

equality checks, but int and bool are not).

x != y Returns false if x and y are equal. Incompatible types return true.

x <> y Alias of [x != y]

x > y Returns true if x is more than y, otherwise returns false.

x < y Returns true if x is less than y, otherwise returns false.

x >= y Returns true if x is more than or equal to y, otherwise returns false.

x <= y Returns true if x is less than or equal to y, otherwise returns false.

[EXPRESSION] Ternary operator expressions:

b ? x : y Returns x if b evaluates to true, otherwise returns y.

Because variables are weak-typed, you are allowed to have x and y as different types.

[EXPRESSION] Function call expressions

f() Calls function f with 0 parameter

f(x) Calls function f with 1 parameter: x

f(x, y) Calls function f with 2 parameters: x and y. Each function is seperated by a comma (',').

f(g(x), y) Nested function calls. Function g(x) will be evaluated before passing the result to f()

All functions accept any number of parameters. However, most functions perform checks and throw errors

if wrong types or wrong parameter numbers are supplied. Check the documentation for each function for

details.

[STATEMENT] Statement syntax

x = EXPRESSION Simple assignment of the result of EXPRESSION to x

EXPRESSION An expression. The output from this expression is disregarded.

[STATEMENT] Assignment operator statements

x += y Alias of [x = x + y]

x -= y Alias of [x = x - y]

x \*= y Alias of [x = x \* y]

x /= y Alias of [x = x / y]

x %= y Alias of [x = x % y]

x |= y Alias of [x = x || y]

x &= y Alias of [x = x && y]

[STATEMENT] Multiple statement blocks

if-then-else block

Syntax:

if (EXPRESSION\_BOOL) then {LIST\_OF\_STATEMENTS}

or if (EXPRESSION\_BOOL) then {LIST\_OF\_STATEMENTS} else {LIST\_OF\_STATEMENTS}

Evaluates LIST\_OF\_STATEMENTS in the then block if EXPRESSION\_BOOL evaluates to true.

Otherwise, evaluates LIST\_OF\_STATEMENTS in the else block, if available

Context:

EXPRESSION\_BOOL An EXPRESSION that returns a bool

LIST\_OF\_STATEMENTS A list of statements (each statement ending with (';'))

If only one statement is used, the encapsulating braces ('{' ... '}') is optional.

foreach-in block

Syntax:

foreach(VARIABLE in VARIABLE\_LIST) {LIST\_OF\_STATEMENTS}

For each value in VARIABLE\_LIST, assign this value to VARIABLE, then evaluate LIST\_OF\_STATEMENTS.

Context:

VARIABLE A variable used to contain each member of VARIABLE\_LIST

VARIABLE\_LIST A variable assigned to an array structure.

Array structures are currently not yet supported by literals. However, they can be returned

from a function.

LIST\_OF\_STATEMENTS A list of statements (each statement ending with (';'))

If only one statement is used, the encapsulating braces ('{' ... '}') is optional.

--------------------------------------------------------------------------------------------------------------------------------

Execution

--------------------------------------------------------------------------------------------------------------------------------

The game, when loading a custom scenario, will load script files defined in its Scenario file.

It will then execute the following functions in order:

- The script whose name is defined under Fn\_loadfaction. If a script by this name is not found, this step is skipped.

- The script whose name is defined under Fn\_loadscene. If a script by this name is not found, this step is skipped.

- The script whose name is defined under Fn\_load. If a script by this name is not found, this step is skipped.

When attempting to generate a player (at initialization or after player death), it will execute the following:

- The script whose name is defined under Fn\_makeplayer. If a script by this name is not found, this step is skipped.

Every game tick, the game will execute the following:

- The scripts whose names are defined under Fns\_gametick.

In your script, you may link the execution of other scripts via one of two functions:

- CallScript function

CallScript("scriptname");

This jumps execution into the script "scriptname".

- AddEvent function

AddEvent(1.0, "scriptname");

This defers execution of script "scriptname" to 1.0 in-game seconds after the current time.

--------------------------------------------------------------------------------------------------------------------------------

Functions

--------------------------------------------------------------------------------------------------------------------------------

**Context Functions**

Context functions are the link between script code to in-game functions. These functions allow you to access parts of the game operation necessary for you build your own scenario.

**Format**

Context functions are called with the following format

Parameterless null function  
 Context function contract: **void Scene.FadeOut()** Example call in script: **Scene.FadeOut();**

Parameterized null function  
 Context function contract: **void Scene.SetMaxBounds(float3 value)** Example call in script: **Scene.SetMaxBounds({10000,200,30000});**

Parameterless value function  
 Context function contract: **int Player.GetActor()** Example call in script: **int playerID = Player.GetActor();**

Parameterized value function  
 Context function contract: **string Actor.GetActorType(int actorID)** Example call in script: **string type = Actor.GetActorType(playerID);**

**void** functions will return NULL, which hold no intrinsic value and cannot be operated with other values except equality / inequality checks. Other functions may return a value depending on the function’s purpose.

**Scripting**

**bool Script.TryCall(string script\_name)**

Attempts to call the script function with the name **script\_name**. If the script is not found, returns **false**.

**bool Script.Call(string script\_name)**

Transfers execution to the script function with the name **script\_name**. If the script is not found, a runtime error will be thrown.

Error: Attempted to call non-existent script ‘<script\_name>’

**Scene**

**void Scene.SetMaxBounds(float3 value)**

Sets the coordinate of the upper bound of the scenario boundary to this **value**. The scenario boundary limits the space where the player can travel in.

**void Scene.SetMinBounds(float3 value)**

Sets the coordinate of the lower bound of the scenario boundary to this **value**. The scenario boundary limits the space where the player can travel in.

**void Scene.SetMaxAIBounds(float3 value)**

Sets the coordinate of the upper bound of the AI boundary to this **value**. The AI will attempt to stay in the space within the boundary limits.

**void Scene.SetMinAIBounds(float3 value)**

Sets the coordinate of the lower bound of the AI boundary to this **value**. The AI will attempt to stay in the space within the boundary limits.

**void Scene.FadeOut()**

Initiate the fade out sequence. After fading out, the game checks if the GameWon game state is enabled. If yes, the scenario diverts to the Game Over menu, otherwise the game checks whether the GameWon game state is enabled. If yes, the scenario diverts to the Game Won menu, otherwise the game will fade in as normal.

**Camera**

**void Camera.SetPlayerLook()**

Sets the camera to use the player vision. (Follow the player craft)

**void Camera.SetSceneLook()**

Sets the camera to use the scene vision. Set the position of the camera and its target using Camera.SetSceneLook\_LookAtX and Camera.SetSceneLook\_LookFromX functions.

**void Camera.SetDeathLook()**

Sets the camera to use the death vision. (Circle the player craft)

**void Camera.EnableFreeLook(bool enabled)**

Enables / Disables free vision. When enabled, this removes player mouse control from the player craft. Instead, the player may use the mouse to rotate the camera around the craft.

**void Camera.SetSceneLook\_LookAtActor(int actorID)  
void Camera.SetSceneLook\_LookAtActor(int actorID, float3 offsetXYZ)  
void Camera.SetSceneLook\_LookAtActor(int actorID, float3 offsetXYZ, float3 offsetRelative)**

Sets the scene camera target to the actor with the given **actorID**. If no such actor is found, do nothing.

If provided, **offsetXYZ** and/or **offsetRelative** determine world and relative offset positions from the actor body respectively.

**void Camera.SetSceneLook\_LookAtPoint(float3 point)**

Sets the scene camera target to a fixed **point**.

**void Camera.SetSceneLook\_LookFromActor(int actorID)  
void Camera.SetSceneLook\_LookFromActor(int actorID, float3 offsetXYZ)  
void Camera.SetSceneLook\_LookFromActor(int actorID, float3 offsetXYZ, float3 offsetRelative)**

Sets the scene camera position to the actor with the given **actorID**. If no such actor is found, do nothing.

If provided, **offsetXYZ** and/or **offsetRelative** determine world and relative offset positions from the actor body respectively.

**void Camera.SetSceneLook\_LookAtPoint(float3 point)**

Sets the scene camera position to a fixed **point**.

**Squad**

**int[] Squad.Spawn(string actorType, string squadName, string faction, int count, float spawnDelay, bool entryByHyperspace, float3 position, float3 rotation, string formation, float formationSpacing, float aiWaitDelay, string huntTargetType, string[] registries)**

Spawns a squad of actors of this **actorType**, owned by this **faction**. The squad will have **count** number of members, will bear designations **squadName** followed by a number from 1 up to **count**. The squad will spawn in **spawnDelay** in-game seconds after this function call, at the specified **position** and **rotation**. If **entryByHyperspace** is enabled, the spawned craft will hyperspace in using **rotation** as the entry vector. Otherwise, the craft simply appears. All actors will be placed in **formation** with a spacing distance defined by **formationSpacing**. All actors will be in a squad with the first actor as the squad leader. All actors in the squad will register themselves in each group specified in **registries**.

After spawning in, the AI governing actors will first wait for **aiWaitDelay** seconds, then proceed to hunt the targets that matches the type given by **huntTargetType**.

This function returns an **int[]** with the actorIDs of the spawned actors.

**bool Squad.AddToSquad(int actorID, int actor2ID)**

The actor of this **actor2ID** will join the squad with actor **actorID**. If neither actor exists, do nothing and return **false**. Otherwise, returns **true**.

**bool Squad.RemoveFromSquad(int actorID)**

The actor of this **actorID** will be removed from any existing squad it is in. If the actor does not exist, do nothing and return **false**. Otherwise, returns **true**.

**bool Squad.MakeSquadLeader(int actorID)**

The actor of this **actorID** will be made the squad leader (hence first member) of its current squad. If the actor does not exist, do nothing and return **false**. Otherwise, returns **true**.

**Actor**

**int Actor.Spawn(string actorType, string name, string faction, string sidebarName, float spawnDelay, float3 position, float3 rotation, string[] registries)**

Spawns an actor of this **actorType**, owned by this **faction**. The actor will bear a designation **name**. If the actor appears on the sidebar, the display name **sidebarName** will be used. The actor will spawn in **spawnDelay** in-game seconds after this function call, at the specified **position** and **rotation**. The actor will register itself in each group specified in **registries**.

This function returns an **int** with the actorID of the spawned actor.

**void Actor.QueueAtSpawner(int actorID, int spawnerID)**

Removes an actor of this **actorID** from the world and place it in the spawn queue of an actor of this **spawnerID**. If neither actor exists, do nothing. Note that the spawner should have a hangar or equivalent spawner add-on if this actor is to be spawned into the world.

**string Actor.GetActorType(int actorID)**

Returns the actor type ID of an actor by this **actorID**. If the actor does not exist, return an empty string **“”**.

**bool Actor.IsFighter(int actorID)**

Returns whether the actor with this **actorID** is a fighter. An actor is positively identified as a fighter if its TargetType definition includes the FIGHTER flag. If the actor does not exist, return **false**.

**bool Actor.IsLargeShip(int actorID)**

Returns whether the actor with this **actorID** is a large ship. An actor is positively identified as a large ship if its TargetType definition includes the SHIP flag. If the actor does not exist, return **false**.

**bool Actor.IsAlive(int actorID)**

Returns whether the actor with this **actorID** exists on the world and is not in the dead state. Note that the dying state still considered alive by this function.

**float3 Actor.GetLocalPosition(int actorID)**

Returns the local position vector of the actor with this **actorID**. If the actor does not exist, return an empty **float3 {0,0,0}**.

**float3 Actor.GetLocalRotation(int actorID)**

Returns the local rotation vector of the actor with this **actorID**. Rotation is given in degrees. If the actor does not exist, return an empty **float3 {0,0,0}**.

**float3 Actor.GetLocalDirection(int actorID)**

Returns the local direction vector of the actor with this **actorID**. If the actor does not exist, return an empty **float3 {0,0,0}**.

**float3 Actor.GetGlobalPosition(int actorID)**

Returns the world position vector of the actor with this **actorID**. If the actor does not exist, return an empty **float3 {0,0,0}**.

**float3 Actor.GetGlobalRotation(int actorID)**

Returns the world rotation vector of the actor with this **actorID**. Rotation is given in degrees. If the actor does not exist, return an empty **float3 {0,0,0}**.

**float3 Actor.GetGlobalDirection(int actorID)**

Returns the world direction vector of the actor with this **actorID**. If the actor does not exist, return an empty **float3 {0,0,0}**.

**void Actor.SetLocalPosition(int actorID, float3 value)**

Sets the local position vector of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.SetLocalRotation(int actorID, float3 value)**

Sets the local rotation vector of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.SetLocalDirection(int actorID, float3 value)**

Sets the local direction vector of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.LookAtPoint(int actorID, float3 point)**

Sets the rotation vector of the actor with this **actorID** so that it faces the position with the coordinate **point**. The roll (z-rotation) of the actor will be zeroed. If the actor does not exist, do nothing.

**int[] Actor.GetChildren(int actorID)**

Returns an **int[]** with the actorIDs of the children of the actor with this **actorID**. If the actor does not exist, or if the children of the actor has not yet been spawned, return an empty **int[]** **{}**. Note that only children that have been spawned will count. As actors spawn their children after they are spawned in, it is recommended to use this function at least a few frames after requesting their spawn.

**float Actor.GetHP(int actorID)**

Returns the HP (shield + hull ratings) of the actor with this **actorID**. If the actor does not exist, return **0**.

**float Actor.GetShd(int actorID)**

Returns the shield rating of the actor with this **actorID**. If the actor does not exist, return **0**.

**float Actor.GetHull(int actorID)**

Returns the hull rating of the actor with this **actorID**. If the actor does not exist, return **0**.

**float Actor.GetMaxHP(int actorID)**

Returns the maximum HP (shield + hull ratings) of the actor with this **actorID**. If the actor does not exist, return **0**.

**float Actor.GetMaxShd(int actorID)**

Returns the maximum shield rating of the actor with this **actorID**. If the actor does not exist, return **0**.

**float Actor.GetMaxHull(int actorID)**

Returns the maximum hull rating of the actor with this **actorID**. If the actor does not exist, return **0**.

**void Actor.SetHP(int actorID, float value)**

Sets the HP (shield + hull ratings) of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.SetShd(int actorID, float value)**

Sets the shield rating of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.SetHull(int actorID, float value)**

Sets the hull rating of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.SetMaxHP(int actorID, float value)**

Sets the maximum HP (shield + hull ratings) of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.SetMaxShd(int actorID, float value)**

Sets the maximum shield rating of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.SetMaxHull(int actorID, float value)**

Sets the maximum hull rating of the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**float Actor.GetArmor(int actorID, string damageType)**

Returns the damage multiplier on the **damageType** to the actor with this **actorID**. If the actor does not exist, return **0**.

As of version 0.1, the accepted **damageType** values are: **COLLISION**, **LASER**, **LIGHT**, **HEAVY**.

**void Actor.SetArmor(int actorID, string damageType, float value)**

Sets the damage multiplier on the **damageType** to the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

As of version 0.1, the accepted **damageType** values are: **COLLISION**, **LASER**, **LIGHT**, **HEAVY**.

**void Actor.SetArmorAll(int actorID, float value)**

Sets the damage multiplier on all damage types to the actor with this **actorID** to this **value**. If the actor does not exist, do nothing.

**void Actor.RestoreArmor(int actorID)**

Restores the damage multipliers to the actor with this **actorID** back to its actor type definitions. If the actor does not exist, do nothing.

**var Actor.SetProperty(int actorID, string propertyName)  
void Actor.SetProperty(int actorID, string propertyName, var value)**

Gets / Sets a property of the actor with this **actorID**.

As of version 0.1, the following is a list of supported properties:

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| Regen | | |
| Regen.NoRegen | **bool** | If set to **true**, the actor will be immune to regeneration, although it can still apply regeneration to its parent / children / siblings. |
| Regen.Self | **float** | The rate in which an actor replenishes its shield rating per in-game second. |
| Regen.Child | **float** | The rate in which an actor replenishes each of its children’s shield rating per in-game second. |
| Regen.Parent | **float** | The rate in which an actor replenishes its parent’s shield rating per in-game second. |
| Regen.Sibling | **float** | The rate in which an actor replenishes each of its sibling’s shield rating per in-game second. |
| AI | | |
| AI.CanEvade | **bool** | Determines whether the AI governing this actor is allowed to evade upon getting hit. |
| AI.CanRetaliate | **bool** | Determines whether the AI governing this actor is allowed to retaliate upon getting hit. |
| AI.HuntWeight | **float** | Determines the weight for AI hunt calculations. The greater the number, the more probable the actor will be selected as a hunt target. |
| Movement | | |
| Movement.ApplyZBalance | **bool** | Determines whether an actor will self-correct its z-rotation towards zero. |
| Movement.MinSpeed | **float** | Determines the actor’s minimum speed |
| Movement.MaxSpeed | **float** | Determines the actor’s maximum speed |
| Movement.Speed | **float** | Determines the actor’s current speed |
| Movement.MaxSpeedChangeRate | **float** | Determines the maximum rate where the actor can change its speed. |
| Movement.MaxTurnRate | **float** | Determines the maximum rate of turn for the actor |
| Health | | |
| Health.HP | **float** | Determines the actors current HP (shield + hull ratings) |
| Health.Shd | **float** | Determines the actors current shield rating |
| Health.Hull | **float** | Determines the actors current hull rating |
| Health.MaxHP | **float** | Determines the actors maximum HP (shield + hull ratings) |
| Health.MaxShd | **float** | Determines the actors maximum shield rating |
| Health.MaxHull | **float** | Determines the actors maximum hull rating |
| Spawner | | |
| Spawner.Enabled | **float** | Determines whether the actor spawner is enabled. To achieve full functionality of the spawner functions, a hangar-type actor must be assigned as a child of this actor |
| Spawner.SpawnTypes | **string[]** | Determines the possible of actor types to spawn. Each iteration is selected randomly. |
| Spawner.SpawnsRemaining | **int** | Determines the number of spawns remaining. This refers to the number of spawn sets. If the actor’s hangar spawns 4 actors at a time, the effective number of spawned actors is 4 times the number of spawned sets. |
| Transform | | |
| Transform.Scale | **float** | Determines whether an actor will self-correct its z-rotation towards zero. |
| Transform.Position | **float3** | Determines the actor’s local position |
| Transform.Rotation | **float3** | Determines the actor’s local rotation (in degrees) |
| Transform.Direction | **float3** | Determines the actor’s local direction |
| Misc | | |
| InCombat | **bool** | Determines whether the actor is considered a combat object. If set to **false**, it is ignored in AI hunt calculations and aggressive tracking calculations. |
| SideBarName | **string** | Determines the display name of this actor on the side bar, if shown. |