Manual.md - Grip

Manual.md

Manual

Contents

Setup

Game Framework Setup

Real-Time Strategy Plugin for Unreal Engine 4 extends the game framework provided by Unreal Engine with features that are common in real-time strategy games.

For this, we embrace the principle of *composition over inheritance*: Most features are implemented by the means of components to add to your actors, to allow for the most flexibility on your side when building your game, especially when it comes to combining the plugin with other third-party software.

Some things have been carefully added to the existing game framework however, as designed by Epic, and require a few steps to set up.

The plugin also ships with a few assets that are designed to get you started, e.g. with unit AI, but feel free to modify or replace them as you see fit.

Make sure *View Plugin Content* is enabled in your view options.

1. Create a player controller deriving from `RTSPlayerController`.

2. Create a player state deriving from `RTSPlayerState`.
3. Create a game state deriving from `RTSGameState`.
4. Create a HUD deriving from `RTSHUD`.
5. Create a game mode deriving from `RTSGameMode`.
6. Use your player controller, player state, game state, and HUD in your game mode.
7. Create an AI controller deriving from `RTSPawnAIController`.
8. Set the *Pawn Behavior Tree Asset* of the new pawn AI controller to `BT_RTSPawnBehaviorTree`.
9. Set the *Pawn Blackboard Asset* of the new pawn AI controller to `BB_RTSPawnBlackboard`.
10. Set the *Acquisition Object Types* of the new pawn AI controller to `WorldDynamic` (or whichever *Object Type* you're going to use in the collision settings of your units).
11. Create a player start deriving from `RTSPlayerStart`.
12. Create one or more resource types deriving from `RTSResourceType`, setting their names and icons to use in your UI.
13. Add the resource types to the `PlayerResourcesComponent` of your player controller.

Camera Setup

Usually, players control a single pawn in Unreal Engine. However, in the case of real-time strategy games, players control many units from a camera perspective far out. Thus, the plugin works best when using a simple pawn with a normal camera whose location reflects what the player wants to see right now. Individual units are not directly possessed by the player controllers, but just "owned" by them.

Creating The Camera

1. Create a pawn blueprint.
2. Add a `Camera` component.
3. Set the *Location* of the `Camera` component as desired (e.g. X = 0, Y = 0, Z = 1500).
4. Set the *Rotation* of the `Camera` component as desired (e.g. X = 0, Y = -75, Z = 0).
5. Use the pawn as *Default Pawn Class* in your game mode.

Setting Up Camera Movement

1. Bind the axis `MoveCameraLeftRight` (e.g. to Left and Right keys).
2. Bind the axis `MoveCameraUpDown` (e.g. to Up and Down keys).
3. Bind the axis `ZoomCamera` (e.g. to the mouse wheel axis).
4. At your `RTSPlayerController`, set the *Camera Speed* (e.g. to 1000).
5. At your `RTSPlayerController`, set the *Camera Scroll Threshold* (e.g. to 20).

6. At your `RTSPlayerController`, set *Camera Zoom Speed*, *Min Camera Distance* and *Max Camera Distance* as desired.
7. At your `RTSPlayerController`, set *Double Group Selection Time* to the number of seconds the player has for rapidly selecting the same control group to center the camera on it.

Input Setup

Many default input actions for real-time strategy games are already provided by the plugin. Given that you use an RTSPlayerController in your game mode, you can bind the following actions which should speak for themselves. Clearly, all of these bindings are optional.

At *Edit > Project Settings > Engine > Input* ...

Selection

1. Bind the action `Select` (e.g. to left mouse button).
2. Bind the actions `SaveControlGroup0` to `SaveControlGroup9` (e.g. to CTRL+0 to CTRL+9).
3. Bind the actions `LoadControlGroup0` to `LoadControlGroup9` (e.g. to 0 to 9).
4. Bind the action `AddSelection` (e.g. to Left Shift).
5. Bind the action `ToggleSelection` (e.g. to Left Ctrl).
6. Bind the action `SelectNextSubgroup` (e.g. to Tab).
7. Bind the action `SelectPreviousSubgroup` (e.g. Shift + Tab).

Camera

1. Bind the actions `SaveCameraLocation0` to `SaveCameraLocation4` (e.g. to CTRL+F5 to CTRL+F9).
2. Bind the actions `LoadCameraLocation0` to `LoadCameraLocation9` (e.g. to F5 to F9).

Orders

1. Bind the action `IssueOrder` (e.g. to the right mouse button). This will enable typical smart orders, such as moving when right-clicking ground, and attacking when right-clicking enemies.
2. Bind the action `IssueStopOrder` (e.g. to the S key).

Health Bars

1. Bind the action `ShowHealthBars` (e.g. to the LeftAlt key).

Production

1. Bind the action `CancelProduction` (e.g. to Escape).

2. Bind the action `ShowProductionProgressBars` (e.g. to the LeftAlt key).

Construction

1. Bind the action `ConfirmBuildingPlacement` (e.g. to Left Mouse Button).
2. Bind the action `CancelBuildingPlacement` (e.g. to Right Mouse Button).
3. Bind the action `CancelConstruction` (e.g. to Escape).
4. Bind the action `ShowConstructionProgressBars` (e.g. to the LeftAlt key).

Gameplay Tags

The plugin makes use of gameplay tags for enabling condition-based gameplay (such as whether a unit can be attacked or not).

At *Edit > Project Settings > Project > Gameplay Tags*, add `DT_RTSGameplayTags` to the *Gameplay Tag Table List*.

Game Modes

Clearly, even for real-time strategy games, you may want to define multiple game modes (such as classic Skirmish or story-based campaigns). Thus, we won't restrict you too much here, but just want to provide a few basic things that you might find useful.

Initialization

1. Set the *Initial Actors* and their locations for your game mode. This will spawn initial units for each player at their player start as soon as the game starts.

Teams

1. Set *Num Teams* to the number of teams your game mode supports.

Game Over

1. Optionally, set the *Defeat Condition Actor Classes* for your `RTSGameMode`. This will check whether any actors of the specified types exist for a player whenever he or she loses a unit. If no actor of the specified type remains, the player is defeated.

In that case, the game mode will raise the OnPlayerDefeated event to be overridden in subclasses (either blueprint or C++). Note that it is up to you to define how defeated players should be handled, and if/when the game is over, e.g. whether you've making a 1v1, FFA or team game.

Creating Maps

For the plugin, you'll design and create your maps the same way you're used to when using Unreal Engine, for the most part. This section serves as a short checklist for you, and highlights some setup that is supposed to make it easier for you to get started. Some steps are mandatory for some features of the plugin to work, however, such as vision.

Game Mode & Geometry

1. Use your game mode in the world settings.
2. Create your level geometry and lighting as usual.

Camera

1. Add an `RTSCameraBoundsVolume` to the map.
2. Use the *Brush Settings* to adjust the camera bounds as desired.

Navigation

1. Add a `NavMeshBoundsVolume` to the map.
2. Use the *Brush Settings* to adjust have the nav mesh bounds encompass your whole level.
3. Build navigation. You may press P to review your results in the viewport.

Player Starts

1. Add `RTSPlayerStart`s to the map.
2. Set the *Team Index* for each player start.

Minimap

1. Add an `RTSMinimapVolume` to the very center of your map.
2. Set its brush size to match the extents of your playable map.
3. Set the *Minimap Image* to a nice top-down screenshot of your map.

Fog Of War

1. Add an `RTSVisionVolume` to the very center of your map, encompassing the whole valid visible map area.
2. Set the *Size In Tiles* of the vision volume to match your minimap background images (e.g. 256).
3. Set the *Tile Height* of the vision volume to the height of a single height level of your map, in cm (e.g. 250).
4. Set the *Height Level Trace Channel* if you want special geometry to affect your height levels, only.
5. Add a `PostProcessVolume` to your map, and check *Infinite Extent (Unbound)*.

6. Add an `RTSFogOfWarActor` to your map.
7. Set the *Fog Of War Volume* reference to the post process volume created before.
8. Set the *Fog Of War Material* of the actor (e.g. to the `M_RTSFogOfWar` material shipped with the plugin).

Pre-Placed Units

1. Add any actors that should initially on the battlefield.
2. For each of these actors, at the `RTSOwnerComponent`, set the *Initial Owner Player Index* to specify which player should own them.
3. When pre-placing buildings, at the `RTSConstructionSiteComponent`, set their *State* to *Finished* if they should be ready from the beginning.

Creating Units

As mentioned before, most features of the plugin are implemented by the means of components to add to your actors. Thus, for adding new units (or buildings), you'll usually create a pawn or character, and add various components, depending on the capabitilies you want the unit to have. This approach enables you to combine features just as you like, for instance having buildings that can attack or units that resources can be returned to.

Appearance

1. Create a new pawn or character blueprint.
2. Check the *Replicates* flag.
3. Add a static or skeletal mesh and setup its location, rotation and scale as usual.
4. Setup collision (e.g. Capsule Collision) as usual. You may want to disable the collision of your mesh and rely on its capsule instead.
5. Setup your animations. (If you're new to the Unreal animation system, we can readily recommend the tutorial at https://docs.unrealengine.com/latest/INT/Programming/Tutorials/FirstPersonShooter/4/index.html)
6. Add an `RTSNameComponent` and set its localized name if you want to show it in any kind of ingame UI.
7. Add an `RTSDescriptionComponent` and set its localized text if you want to show it in any kind of ingame UI.
8. Add an `RTSPortraitComponent` and set its portrait if you want to show it in any kind of ingame UI.
9. Add an `RTSSelectableComponent`, and set its selection circle material (e.g. to `M_RTSSelectionCircle`) and selection sound.
10. If the selection order of your actors matters (e.g. for grouping in your UI), set the *Selection Priority* as well. Untick *Receives Decals* on the static or skeletal mesh to prevent adjacent decals rendering on the unit when in close proximity to other selected units.

11. In case you got special visual effects for highlighting hovered actors, you can listen for the `OnHovered` and `OnUnhovered` events of the component.
12. Add an `RTSOwnerComponent`. This will be used to store (and replicate) the owner of the unit for all players (e.g. for showing team colors).
13. Add your `RTSHoveredActorWidgetComponent` (see User Interface).

Movement

1. Add a movement component (e.g. `FloatingPawnMovement` or `CharacterMovement`) and set up its speed properties as usual. The plugin also ships with a `RTSPawnMovementComponent` that basically just adds rotation updates to the built-in `FloatingPawnMovement`.

Vision

1. Add the `RTSVision` component to your units and set their *Sight Radius* (e.g. 1000).
2. If your actor should be able to ignore height levels for vision (e.g. watch-towers), check *Ignore Height Levels*.
3. Add a `RTSVisible`component to your actor. That component will manage vibility of that actor, in case multiple effects want to show/hide it (e.g. fog of war, containers).

Combat

AI As mentioned before, units are not directly possessed by player controllers in the plugin. Instead, every unit is possessed by an AIController that will carry out orders issued by the players.

1. Set the pawn AI controller class to your `RTSPawnAIController`.
2. Ensure *Pawn > Auto Possess AI* is set to *Placed in World or Spawned*.

Health & Damage

1. Add the `RTSGameplayTagsComponent` and add the `Status.Permanent.CanBeAttacked` tag to the `RTSGameplayTagsComponent` to any actors that can be attacked.
2. Set the *Maximum Health* of the `RTSHealthComponent`.
3. In case your actor should regenerate health, check *Regenerate Health* and set the *Health Regeneration Rate*.
4. If you want to play animations or visual effects when the actor dies, set *Actor Death Type* to *Stop Gameplay*. In that case, you're responsible of destroying the actor yourself as soon as all of your visual clues have finished playing.
5. Set the *Death Sound* to the sound cue to play when the actor is killed.
6. Add your `RTSHealthBarWidgetComponent` (see User Interface).
7. Add the `RTSAttackComponent` to any actors than can attack.

8. Add an attack to the `RTSAttackComponent` of these actors, setting its *Cooldown, Damage, Range, Acquisition Radius* and *Chase Radius*.

Setting the Damage Type is optional.

Projectiles If you don't specify a projectile, the damage will be applied immediately. In order to setup a projectile for the unit:

1. Create an actor deriving from `RTSProjectile`.
2. Add a static mesh and any visual effects.
3. At the `ProjectileMovement` component, set its *Initial Speed* (e.g. to 1000).
4. If you want your projectile to use a ballistic trajectory, check *Ballistic Trajectory* at the projectile and set the *Ballistic Trajectory Factor* as you like.
5. For dealing area of effect damage, check *Apply Area Of Effect* and set your area of effect radius and collision filters.
6. Set the *Fired Sound* and *Impact Sound* to the sound cues to play when the projectile is fired and detonated, respectively.
7. At the `RTSAttackComponent`, reference the new projectile in your attack.

Production Costs

1. Add the `RTSProductionCostComponent` to everything you want to be produced.
2. Set the *Production Time* for these products.
3. Set the *Resources* to any resources required for production.
4. Set the *Production Cost Type* to to *Pay Immediately* if all costs should be paid in full when starting production, or to *Pay Over Time* for continuously paying costs (similar to Command & Conquer).
5. Set the *Refund Factor* to the factor to multiply refunded resources with after cancelation.
6. Set the *Finished Sound* to the sound cue to play when the actor spaws after being produced.
7. Add the `RTSRequirementsComponent` if the actor should have any requirements, and set the *Required Actors*.

Construction

1. Add an `RTSBuilderComponent` to any actors you want to be able to construct buildings.
2. Set the *Constructible Building Classes* for these builders.
3. Check *Enter Construction Site* if you want the builder to be unavailable while building (similar to Orcs in WarCraft). In that case, add a `RTSContainableComponent` as well.

Gathering

1. Add an `RTSGathererComponent` to any actor that should be able to gather resources.
2. Add any resource type the gatherer should be able to gather to *Gathered Resources*.
   (a) Gathering works similar to attacks, with "damage" and "cooldown". Set *Amount Per Gathering* to the value to add to the gatherers inventory each time the cooldown is finished.
   (b) Set the *Cooldown* to the time between two gatherings.
   (c) Set the *Capacity* to the amount of resources the gatherer can carry before returning to a resource drain.
   (d) Check *Needs Return To Drain* if the gatherer needs to move to another actor for returning resources (e.g. Age of Empires). Uncheck if they should return all gathered resources immediately when hitting the capacity limit (e.g. Undead in WarCraft).
   (e) Set *Range* as desired.
3. Add all *Resource Source Actor Classes* the gatherer may gather from (e.g. Undead in Warcraft need Haunted Gold Mine).
4. Set the *Resource Sweep Radius* to the radius in which the gatherer should look for similar resources if their current source is depleted.

Bounties

1. Add an `RTSBountyComponent` to any actors you want to grant bounties to killing players.
2. Set the *Bounty* for these actors.

Multiplayer

1. If you're working on an online game, check *Always Relevant* for your actor.

Creating Buildings

From the perspective of the plugin, buildings are just units with a few additional components. There's no special class for buildings; their setup has just been moved to this manual section because that many people would explicitly look for that. In fact, you can mix and match the setup outlined in this section with all of the other sections. This allows you to create truly deep gameplay, such as units that serve as resource sources, or produce other units.

Construction

1. See Creating Units (Appearance, Health & Damage, Projectiles).
2. Add an `RTSConstructionSiteComponent` and set the *Construction Time*.
3. Set the *Construction Costs* to any resources required for construction.

4. Set the *Construction Cost Type* to to *Pay Immediately* if all costs should be paid in full when starting construction, or to *Pay Over Time* for continuously paying costs (similar to Command & Conquer).
5. Set the *Refund Factor* to the factor to multiply refunded resources with after cancelation.
6. Set the *Consumes Builders* flag if builders working at the construction site should be destroyed when finished (similar to Zerg in StarCraft).
7. Set *Max Assigned Builders* if you want to require a builder to work at the construction site to make progress, and/or to allow multi-building (similar to Age of Empires).
8. Set the *Progress Made Automatically* and *Progress Made Per Builder* factors.
9. Set *Initial Health Percentage* to a value between 0 and 1 to specify how much health the construction site should start with.
10. Set the *Start Immediately* flag unless you want to trigger construction start from script.
11. If you want to use grid-based building placement, set the *Grid Width and Height*.
12. Set the *Finished Sound* to the sound cue to play when the construction is finished.
13. Add an `RTSContainerComponent` if you want builders to enter the building site while building. Its capacity value will be automatically set at runtime to reflect *Max Assigned Builders* of the construction site.
14. Add your `RTSConstructionProgressBarWidgetComponent` (see User Interface).

Production

1. Add an `RTSProductionComponent` to any actors you want to be able to produce units or research technology.
2. Add everything you want to produce or research to the *Available Products* for these factories.
3. Set the *Queue Count*, specifying how many products can be produced in parallel.
4. Set the *Capacity Per Queue*, specifying how many products can be produced one after another.
5. Add your `RTSProductionProgressBarWidgetComponent` (see User Interface).

Note that, technically, producing units does not differ from researching technology. You can create actor blueprints without physical representation for each technology to research, and add them as products. Then, you can check whether any player owns an actor of that technology for checking a tech tree.

Resource Drain

1. Add an `RTSResourceDrainComponent` for each type of building gatherers may return resources to.
2. Set the resource types to all resources accepted by the drain.

Defense

1. If your building has an `RTSAttackComponent`, check *Preview Attack Range* if you want the attack range of your building to be previewed while placing the building.

Vision

1. At the `RTSVisible`component of your building, check *Don't Hide After Seen* if you want your building to stay visible even through for of war.

Projectile Impacts

1. While your building is under attack, you might want to prevent every single projectile hitting the exact same location. You can add a `RTSProjectileTargetComponent` to your actor, and specify the *Target Sockets* to have projectile fly towards. These sockets have to be added to your static mesh using the built-in Unreal socket manager.
2. If your building has multiple mesh components, add a *component tag* to the desired mesh component and specify the same tag at your `RTSProjectileTargetComponent`.

Creating Resource Sources

In case you missed that step earlier, make your to set up your resource types as explained in Setup. Then, create resource sources as follows:

1. See Creating Units (Appearance only; can be a standard actor).
2. Add an `RTSResourceSourceComponent`.
3. Set the resource type and maximum and current resources of the source.
4. Set the gathering factor for increasing the yield of any gatherers (e.g. golden minerals in StarCraft).
5. If you want gatherers to enter the resource source (e.g. Refinery in StarCraft), check *Gatherer Must Enter*, set the *Gatherer Capacity*, and add an `RTSContainerComponent`. In that case, add a `RTSContainableComponent` to all gatherers as well.

Creating The User Interface

Usually, you'll want to create a very individual user interface for your own game. However, some things are very common to real-time strategy games, such as health bars or minimaps, and we want to provide you a small head start at least,

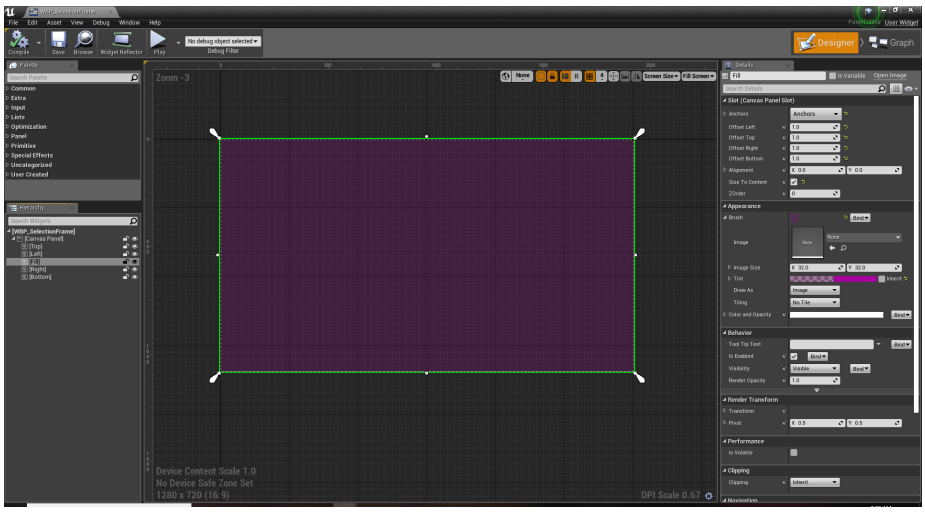
mostly by the means of events you can implement. As always, feel free to create your own UI widgets as you see fit - you should be able to apply them easily with the plugin.

Selection Frames

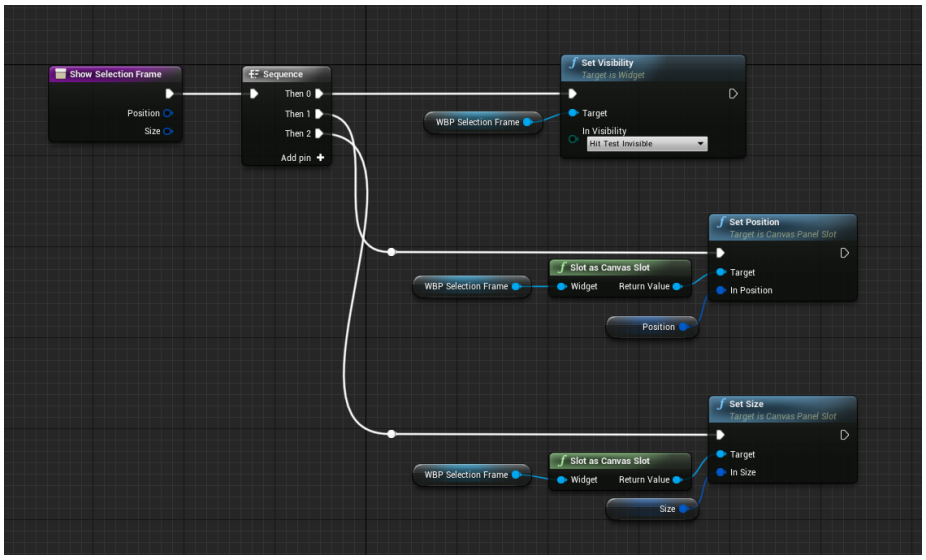
In your HUD, implement the `DrawSelectionFrame` and `HideSelectionFrame` events as desired.

Example:

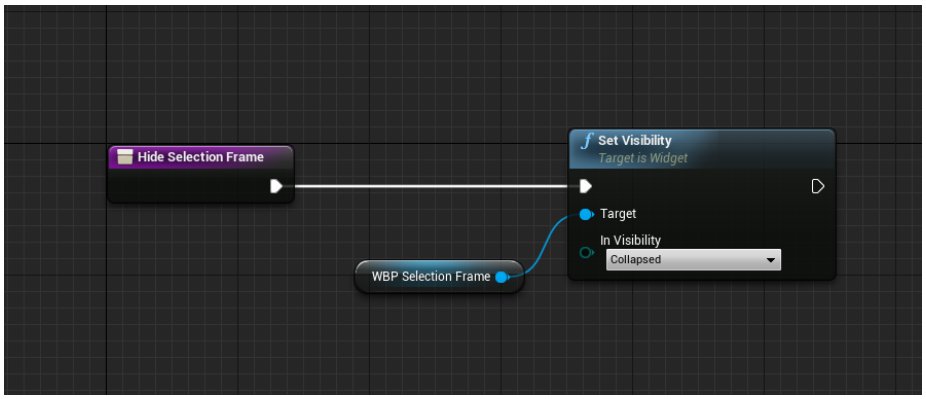
1. Create a widget for drawing the selection frame.



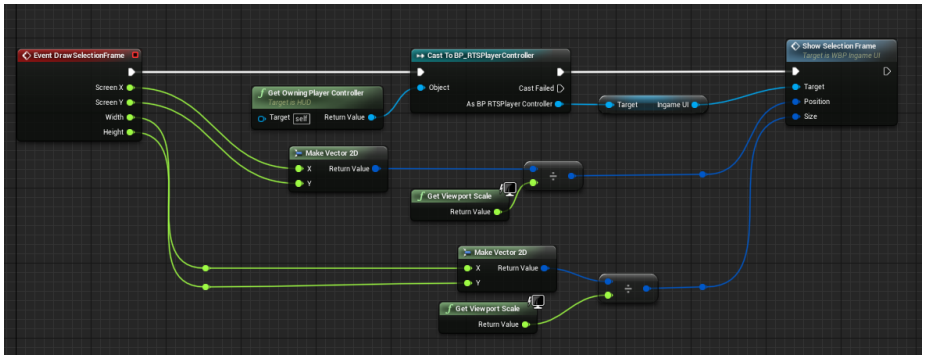
2. Add your widget to any kind of user interface your player controller knows about.
3. In that user interface, provide a function for showing the selection frame.



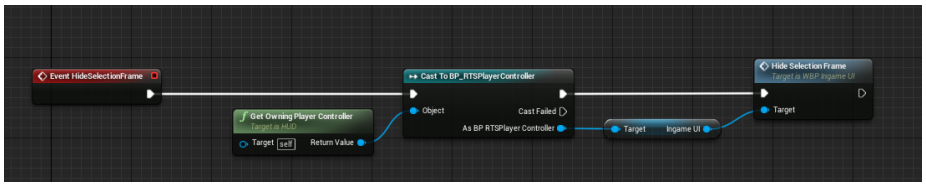
4. In the user interface, provide a function for hiding the selection frame.



5. In your HUD, forward the `DrawSelectionFrame` event to your UI.



6. In the HUD, forward the `HideSelectionFrame` event to your UI.

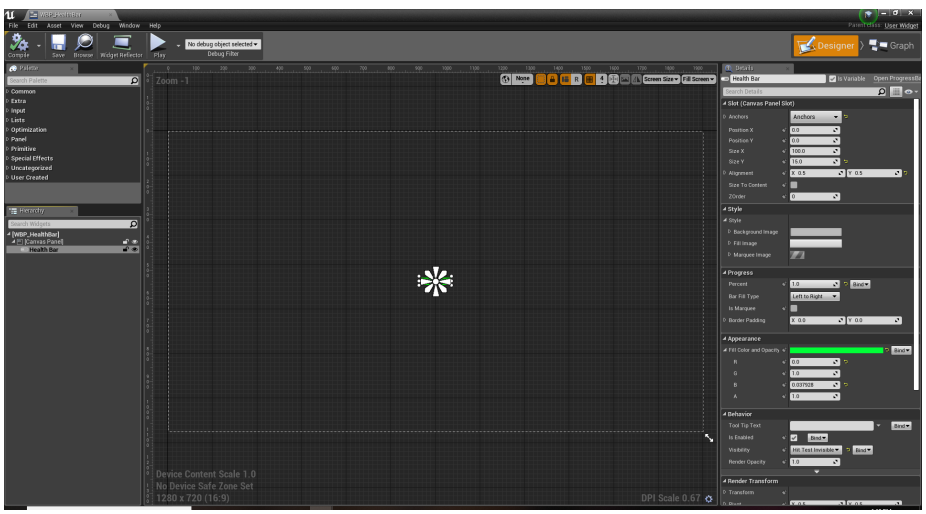


Selected Unit Status

1. Create a new widget blueprint.
2. Create the widget where appropriate (e.g. `BeginPlay` of your player controller) and add it to your viewport.
3. Listen to the `OnSelectionChanged` event broadcasted by the `RTSPlayerController` and update your UI.

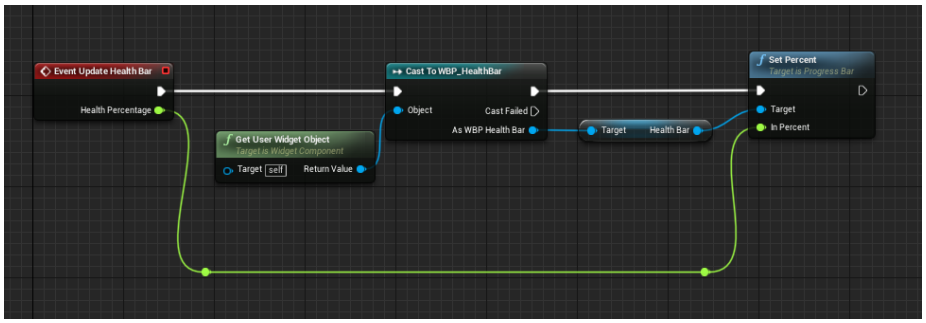
Health Bars

1. In your HUD, set *Always Show Health Bars, Show Hover Health Bars, Show Selection Health Bars* and *Show Hotkey Health Bars* as desired.
2. Create a widget for drawing the health bar.

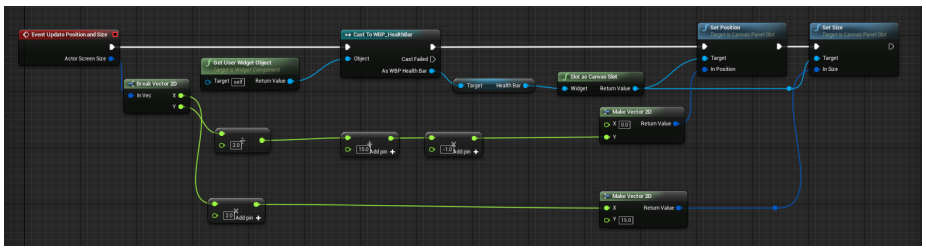


You might want to make sure that the visibility of the widget is set to Hit Test Invisible. Otherwise, it will block mouse input from your player.

3. Create a component deriving from `RTSHealthBarWidgetComponent`, and set its *Widget Class* to your health bar widget.
4. Forward the `UpdateHealthBar` event to your health bar widget.

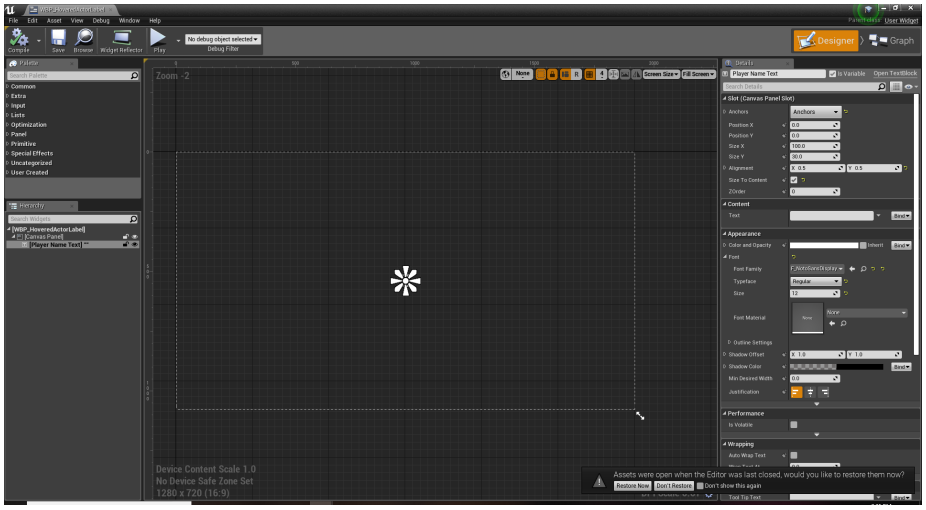


5. Forward the `UpdatePositionAndSize` event to your health bar widget.



Hovered Actors

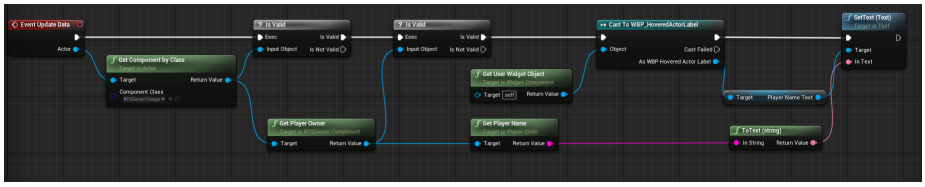
1. Create a widget for drawing name plates (or whatever other information you'd like to display for hovered actors).



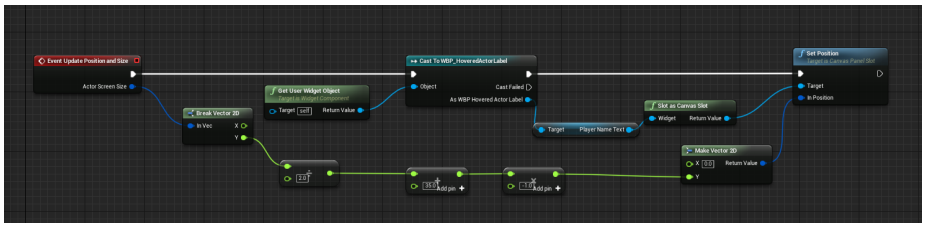
You might want to make sure that the visibility of the widget is set to Hit Test Invisible. Otherwise, it will block mouse input from your player.

2. Create a component deriving from `RTSHoveredActorWidgetComponent`, and set its Widget Class to your new widget widget.

3. Forward the `UpdateData` event to your widget.



4. Forward the `UpdatePositionAndSize` event to your widget.



Building Cursors

1. Create an actor deriving from `RTSBuildingCursor` (or use the `BP_RTSBuildingCursor` shipped with the plugin).
2. If you want to use grid-based building placement, set the *Grid Cell Size* of your building cursor, and set up its collision and navigation check settings as desired.
3. In your player controller, set the building cursor reference.

Note: When having your building cursor query navigation, you need to enable Allow Client Side Navigation in your project settings for this to work in multiplayer.

Range Indicators

1. Create an actor deriving from `RTSRangeIndicator` (or use the `BP_RTSRangeIndicator` shipped with the plugin).
2. At your building cursor, set the range indicator reference.

Production UI

1. Use `GetAvailableProducts` of a selected production actor to create buttons for your production options (e.g. whenever the player controller raises OnSelectionChanged).
2. Call `IssueProductionOrder` of your player controller whenever one of these buttons is clicked.

Production Progress Bars

1. In your HUD, set *Always Show Production Progress Bars, Show Hover Production Progress Bars, Show Selection Production Progress Bars* and *Show Hotkey Production Progress Bars* as desired.
2. Create a widget for drawing the production progress bar.

See the Health Bars section for an example.

3. Create a component deriving from `RTSProductionProgressBarWidgetComponent`, and set its *Widget Class* to your progress bar widget.
4. Forward the `UpdateProductionProgressBar` event to your progress bar widget.
5. Forward the `UpdatePositionAndSize` event to your progress bar widget.

Construction UI

1. Use `GetConstructibleBuildingClasses` of a selected builder to create buttons for your construction options (e.g. whenever the player controller raises `OnSelectionChanged`).
2. Call `BeginBuildingPlacement` of your player controller whenever one of these buttons is clicked.

Construction Progress Bars

1. In your HUD, set *Always Show Construction Progress Bars, Show Hover Construction Progress Bars, Show Selection Construction Progress Bars* and *Show Hotkey Construction Progress Bars* as desired.
2. Create a widget for drawing the construction progress bar.

See the Health Bars section for an example.

3. Create a component deriving from `RTSConstructionProgressBarWidgetComponent`, and set its *Widget Class* to your progress bar widget.
4. Forward the `UpdateConstrutionProgressBar` event to your progress bar widget.
5. Forward the `UpdatePositionAndSize` event to your progress bar widget.

Resources UI

1. Create a widget for showing your current resources.
2. Handle the `OnResourcesChanged` event raised by the `PlayerResourcesComponent` attached to your player controller to update your UI.

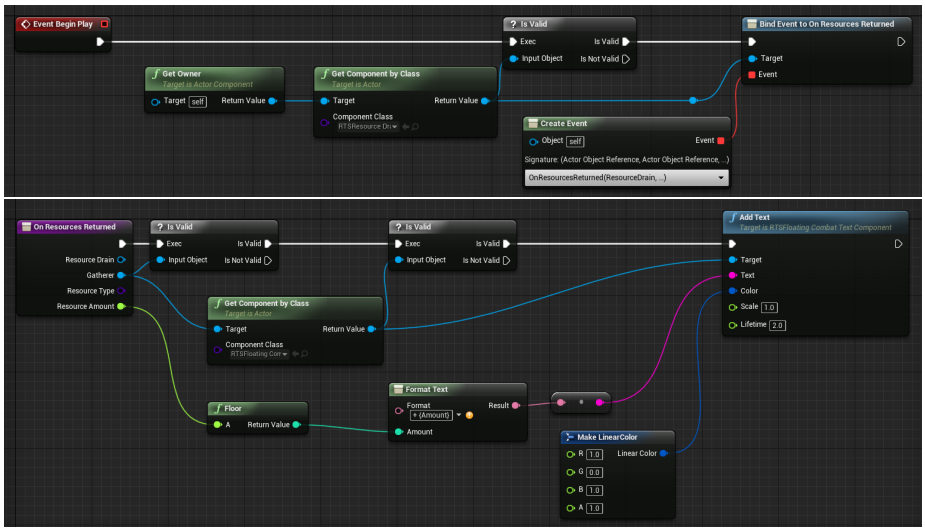
Minimap

1. Add the `WBP_RTSMinimapWidget` to your UI, with a size matching your minimap volume images (e.g. 256 x 256).

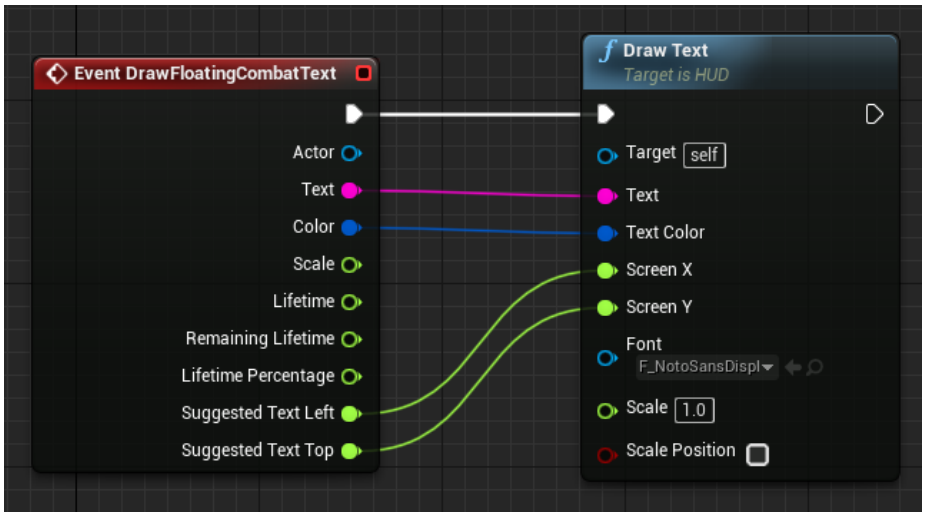
2. Set the *Draw Background, Draw Units With Team Colors, Draw Vision* and *Draw View Frustum* flags as desired.
3. If you checked *Draw Units With Team Colors*, set the *Own Units Brush, Enemy Units Brush* and *Neutral Units Brush* as desired.
4. If you want damaged units to blink on the minimap, set *Damaged Units Blink Brush* and *Damaged Unit Blink Time Seconds* as desired.

Floating Combat Texts

1. In your HUD, enable *Show Floating Combat Texts*.
2. Set *Floating Combat Text Speed* and *Fade Out Floating Combat Texts* as desired.
3. Add a `RTSFloatingCombatTextComponent` to any actor that should be able to display texts above them.
4. Create an actor component for adding the actual floating combat texts.



5. Add your actor component to all actors that should be able to add floating combat texts.
6. In your HUD, handle the `DrawFloatingCombatText` event.



AI Players

The plugin provides basic support for AI players as well. Currently, this doesn't go beyond fulfilling basic build orders, so you'll probably want to extend that, e.g. by determining when to attack other players, and where. At least, this should get you started:

1. Create an AI controller deriving from `RTSPlayerAIController`.
2. Set the *Player Behavior Tree Asset* of your new player AI controller to `BT_RTSPlayerBehaviorTree` (or create your own one).
3. Set the *Player Blackboard Asset* of your new player AI controller to `BB_RTSPlayerBlackboard` (or create your own one).
4. Set up the *Build Order* of your new player AI controller. The AI will produce the specified actors in order, automatically skipping actors that are already available and replacing those that have been destroyed.
5. Set up the *Primary Resource Type* of your new player AI controller. The AI will try and prevent blocking paths between its main building and resource sources of that type.
6. Add your resource types to the `PlayerResourcesComponent` of your player AI controller.
7. Check *Gives Bounty* if killing actors owned by the AI should yield bounties (e.g. for neutral players).
8. Use your player AI controller in your game mode.
9. At your game mode, set *Num AI Players* to the number of AI players you want to spawn.

Scripting

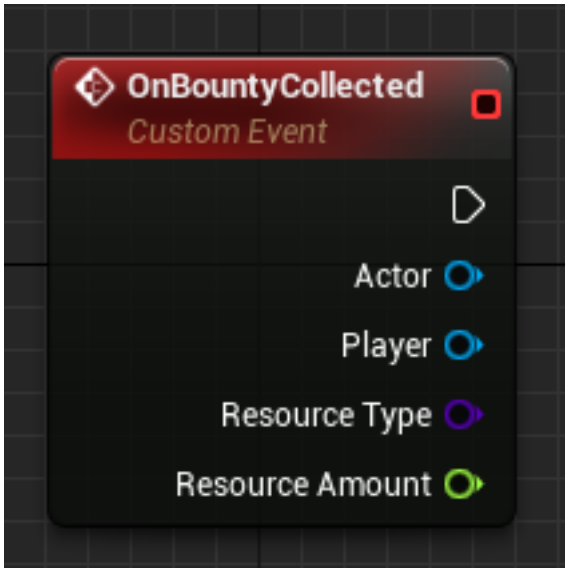
Occasionally, you want to create additional gameplay (especially when creating a story campaign). This section highlights additional functions you can call from blueprints, as well as events you may handle.

Feel free to explore the plugin yourself by looking at what other functions and events each component provides, and open an issue if you're missing something.

Also note that much gameplay of the plugin relies on Gameplay Tags as defined by `Content\Data\DT_RTSGameplayTags.uasset`. This enables you to create unique gameplay and abilities by applying/removing these tags to/from the `URTSGameplayTagsComponent` of your actors. Initially, many of the plugin components apply matching gameplay tags (e.g. `Status.Permanent.CanAttack` from `RTSAttackComponent`). When creating your own actor components, you can derive from `RTSActorComponent` to provide your own *Initial Gameplay Tags* as well. The plugin also provides an own Gameplay Debugger category for checking the gameplay tags of selected actors.

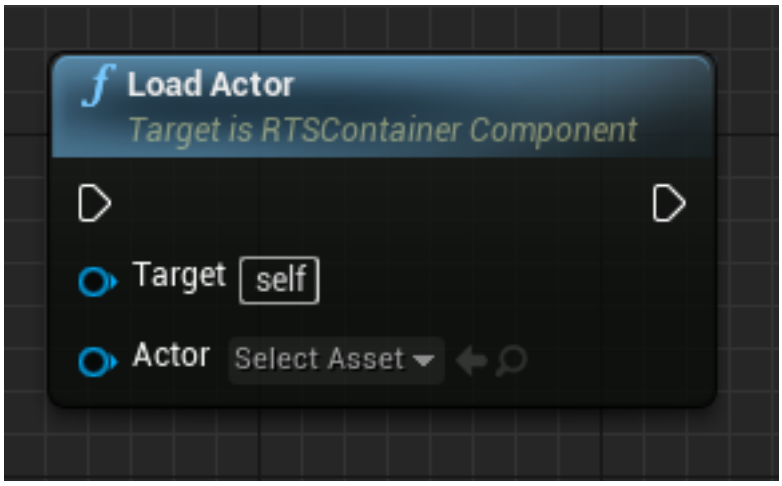
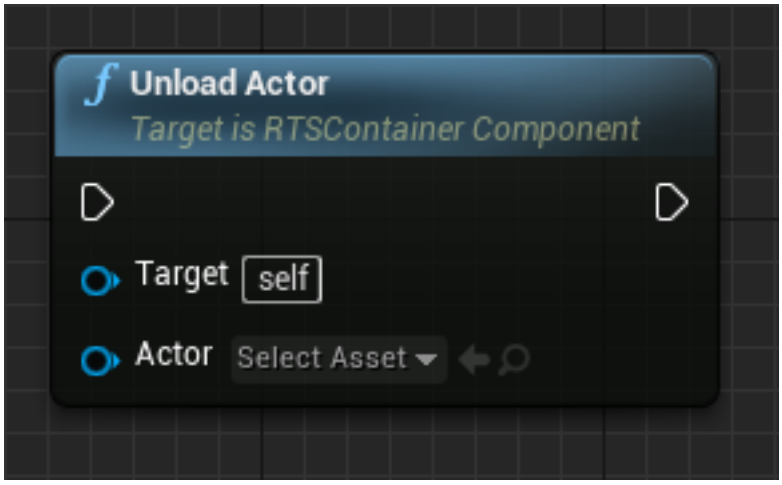
RTSBountyComponent

Events

Event	Description
	Event when the bounty was collected.

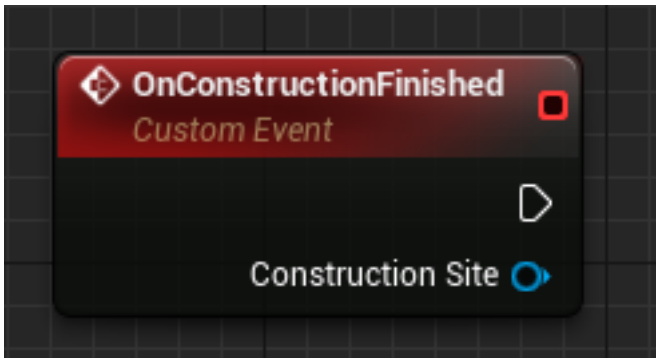
RTSContainerComponent

Functions

Node	Description
	Adds the specified actor to this containe
	Removes the specified actor from this co

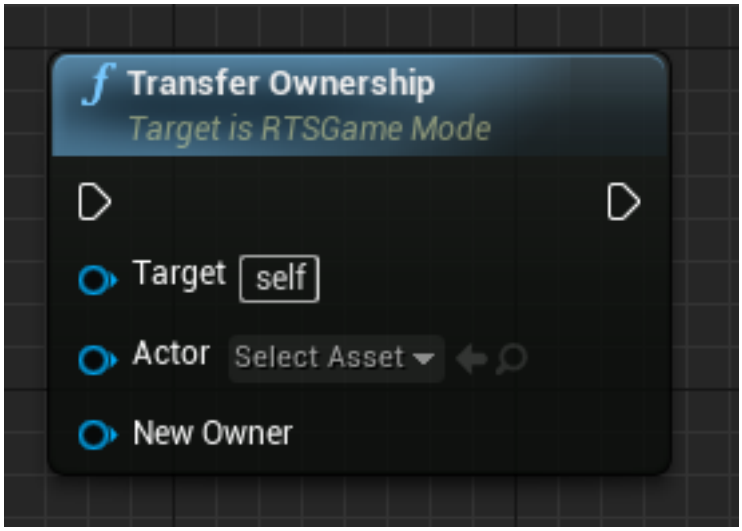
RTSConstructionSiteComponent

Events

Event	Description
	Event when the construction timer has expired.

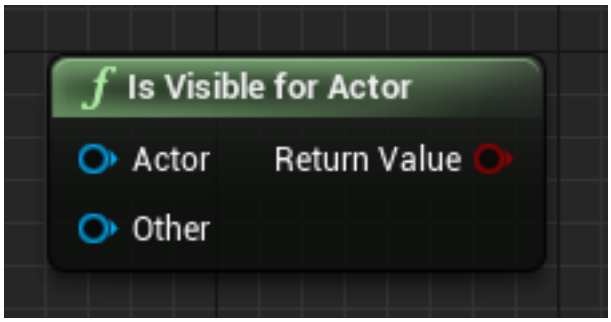
RTSGameMode

Functions

Node	Description
	Sets the specified player as the owner of the

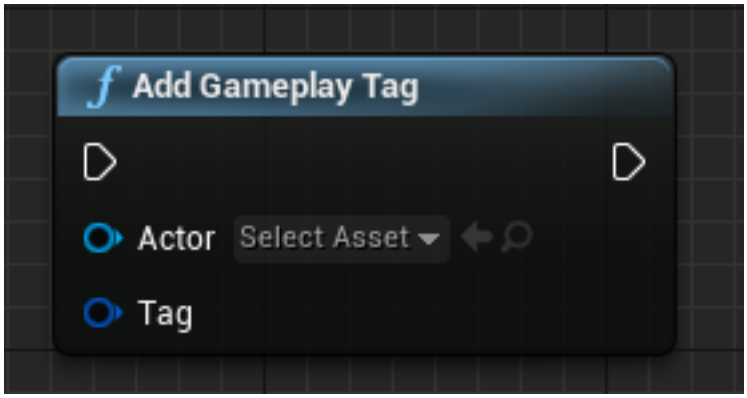
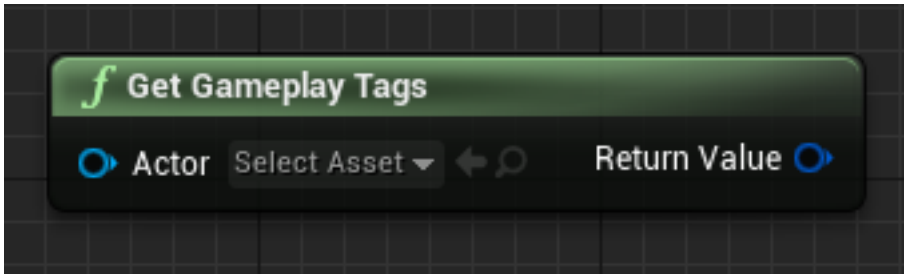
RTSGameplayLibrary

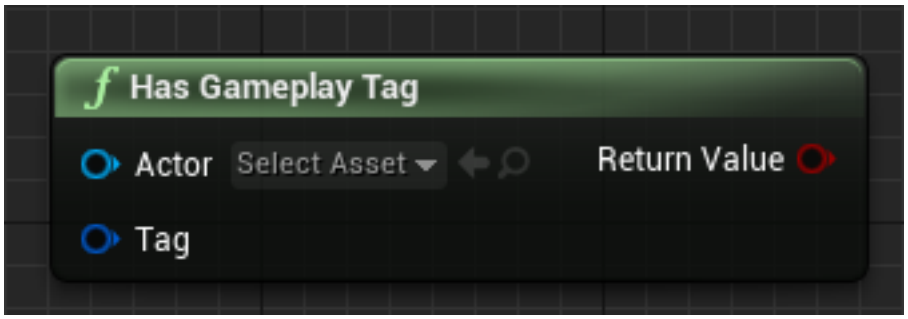
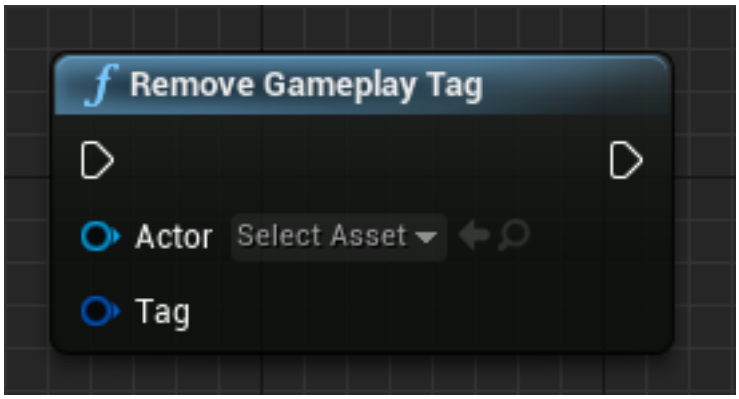
Functions

Node	Description
	Checks whether Other is visible for Actor.

RTSGameplayTagLibrary

Functions

Node	Description
	Applies the passed gameplay
	Gets the current set of gamep

Node	Description
	Checks whether the specified
	Removes the passed gameplay

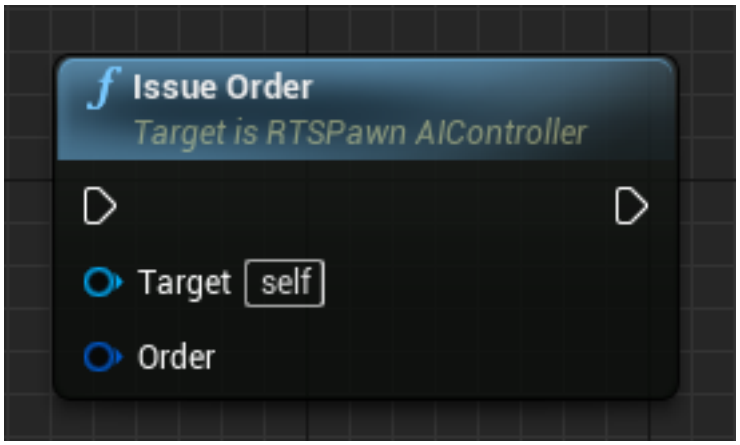
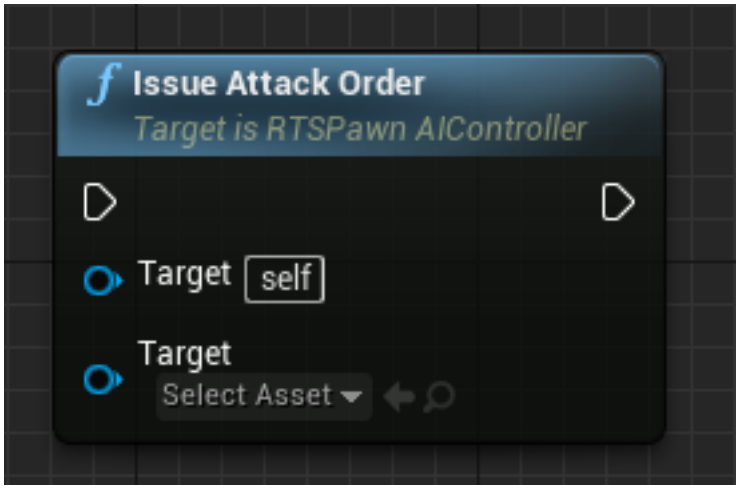
RTSHealthComponent

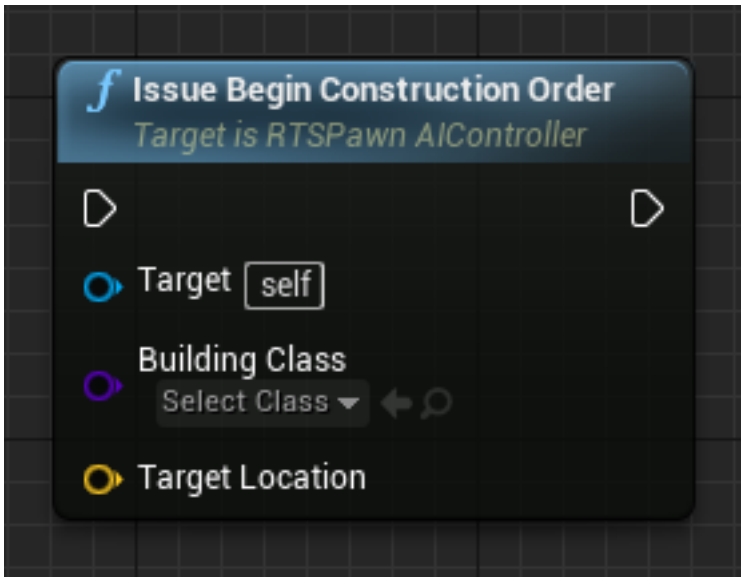
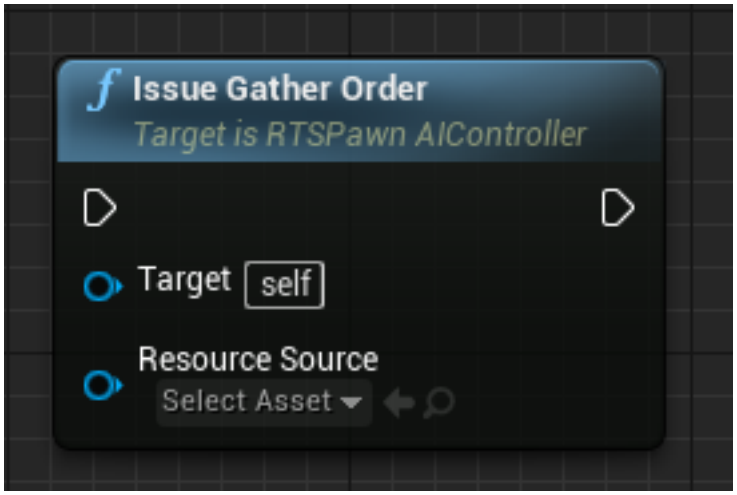
Events

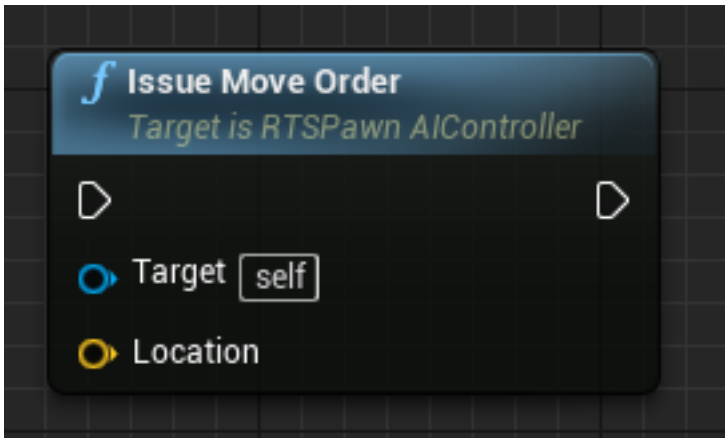
Event	Description
	Event when the current health of the actor has changed.
	Event when the actor has been killed.

RTSPawnAIController

Functions

Node	Description
	Makes the pawn carry out the specified ord
	Makes the pawn attack the specified target.

Node	Description
	Makes the pawn construct the specified buil
	Makes the pawn gather resources from the s

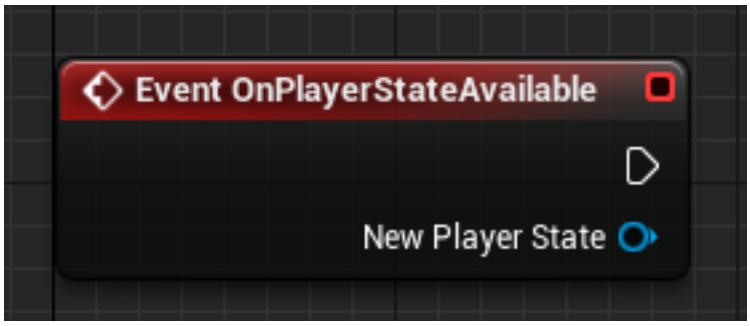
Node	Description
	Makes the pawn move towards the specified
	Makes the pawn stop all actions immediatel

Events

Event	Description
	Event when the pawn has received a new order.

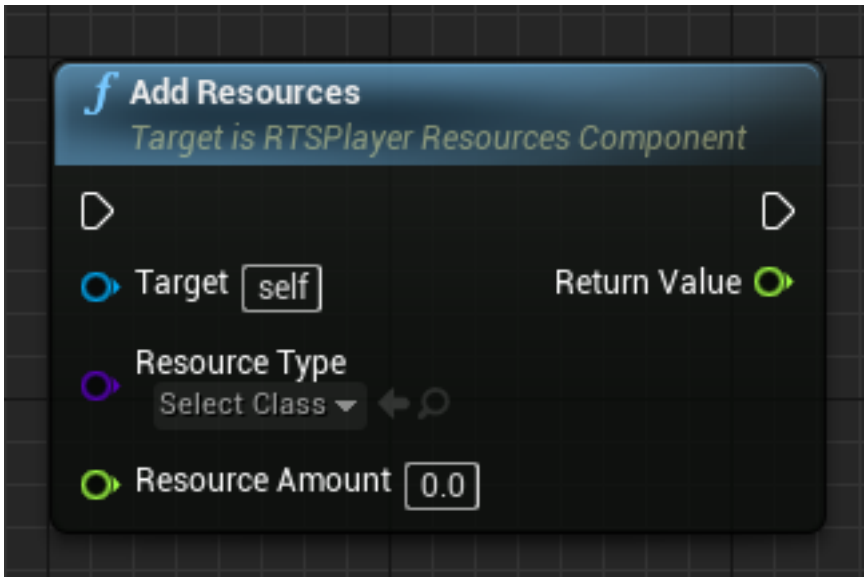
RTSPlayerController

Events

Event	Description
	Event when the player state has been set u

RTSPlayerResourcesComponent

Functions

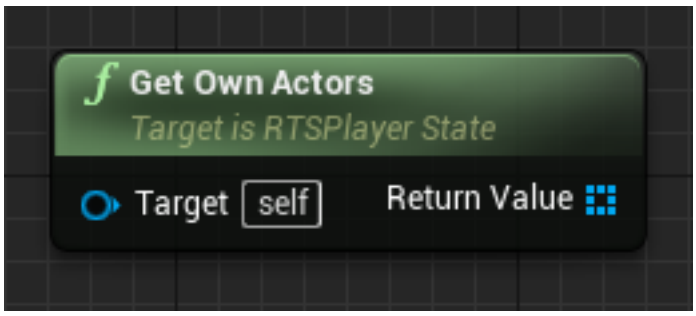
Node	Description
	Adds the specified resources to th

Events

Event	Description
	Event when the current resource stock amount for the

RTSPlayerController

Functions

Node	Description
	Gets the list of actors currently owned by this p

RTSProductionComponent

Events

Event	Description
	Event when the production timer has expired.

Cheats

The plugin comes with a small set of built-in cheats you can use. Feel free to create your own cheat manager and add additional cheats.

1. Create a cheat manager deriving from `RTSCheatManager`.

2. Set the *Resource Types* of your cheat manager.
3. At your `RTSPlayerController`, set the *Cheat Manager* to your cheat manager.
4. At *Edit* > *Project Settings* > *Engine* > *Input*, set and remember your *Console Keys*.

This will unlock the following built-in cheats to use in your console:

Cheat	Description
Boost	Increases construction and production speed.
Damage	Increase damage of all own units.
God	Invulnerability cheat.
NoFog	Toggles fog of war on/off.
Money	Adds resources.
Victory	Defeat all other players.
Weak	Decreases damage of all own units.