GameGraph Readme

You can find the most up to date version of this document at here:

https://docs.google.com/document/d/1glRwCFsv4xCQHawcW4QywXEmoLYDXGcuxpvNLEd_i4Y/edit?usp=sharing

You can find the online API here:

https://dev.inverseclockwork.com/gamegraph/api/

1. Demo materials

1.1. Post-installation notes

All demo materials are contained in GameGraphData and Demo folders. Please delete GameGraphData to have a clean slate for your own work. You can also delete the Demo folder as unnecessary.

1.2. Running the demo in the editor

The system is dependent on addressables and as such “Demo/ICW GameGraph Demo Group” must be contained in the addressable groups. If the addressables aren’t installed before installing the GameGraph package, this has to be dragged into the addressables window manually.
Then load up the BootScene and press play. This scene doesn’t strictly need to be loaded as the system will load regardless (unless specified not to) but this ensures that the staging is more clean.

2. Concepts

1.1. Scene/logic decoupling

As much as is viable, the system is designed to not house logic data on the scene files. This helps with version control and graph calculation.

1.2. State

The system holds a state - that is the savable active condition. This can be serialized and deserialized as needed containing all seed data to reconstruct the game state. It doesn’t contain all runtime information, but enough that it’s kept simple and effective.

1.3. Scene sets

Scenes are loaded from scene sets - which act as the basebones of the states and contain a base scene and then optionally additive scenes and dynamically loaded additive scenes. These dynamic additions are loaded if filter conditions are met.

1.4. Variables

The system has support for simple variable flags.

1.5. Modifiers and modification lists

Scene sets can have modification lists associated with them. The scenes loaded by the set should then have a trigger mechanism for activating that modification list. These modifiers change the variables and load different scene sets.

1.6. Representations

Scenes can have representations of the state by filtering for conditions and running triggers etc. These representations should be resettable and can be changed whichever way while the scene set is loaded.

Representations can be skipped to the end or function as a process. For example, navigation representation can be set upon activation during the same scene set to navigate a character to a point but if the conditions are met and the scene set is loaded after that, to set the character to the appropriate destination from get go.

1.7. Filters

Filters are used by scene set dynamic scenes and by representations to funnel the states during which they're active. These are constructed by simple logic operations and such.

1.8. Graph

For debugging and illustrative purposes, the system can be set to calculate a flow graph of the game states by running through every state and possible modification outside of the play mode. These generated possible states can then be examined and/or loaded for debugging.

1.9. Loading screens

Loading screens can be provided by implementing a LoadingScreen component on a prefab and feeding it in.

3. Workflow with the system

2.1. Generating the prerequisite assets

First, ensure that addressables have a group generated for the system. Then, open "Tools/ICW/GameGraph/SceneSet Viewer" and generate the base assets from there.

2.2. Generating the scene sets

Click the "Create New SceneSet" button and name it. Ensure that the first scene set field is set at some scene set.
Load up the wanted scenes in unity and click "Generate from currently loaded" to attempt an automatic assignment.
You can also manually add the addressable references to the scenes. At least the main scene has to be set.

2.3. Settings

In the "Graph settings" section, the default loading bar can be assigned. One builtin is provided with the package.
First scene set field needs to be set and auto start is generally better to be left on unless you have some other considerations and know that you want to activate the system manually.

2.4. Generating the variables

Open the variable viewer through "Tools/ICW/GameGraph/Variable Viewer" and click "Create a New Variable".
Then select the variable type and rename it by pressing the button with the variable's name on it.
By using "."-character, the variable can be namespaced for organization.

2.5. Setting up the representations in the scenes

Create a game object and add any of the builtin representations to it. For example, spawn representation which is useful for ensuring that if the scene has multiple entry doorways, the player can be spawned by the correct one.
Assign the prefab and configure which previous scene set the spawn is from. Require a previous set toggle being set and the scene set not being assigned is the initial spawn at the beginning of the game.
Events representation can be used to drive Unity Events which is wildly useful.

2.6. Setting up the modifiers in the scenes

Most useful modifier out of the builtin ones is probably the modifier on demand that can be triggered by the system using its API. There is also a modifier on trigger.
Modifiers hook up to a modification list which can be generated from the component. The newly generated modification list is then associated with the scene set and can not be shared between scene sets.
Modifiers can be given requirements that need to be fulfilled before the activation can succeed using a filter.
The actual modifications can be set to change variables or the current scene set which will then trigger a scene load.

2.7. Graph visualization

Open “ICW/Tools/GameGraph/Graph Viewer” and press the “Calculate graph” button.
This generates the graph by running through all possible places in the graph and retrieving the states.
Be warned that the system will cut off recursing on the same scene set to prevent infinite looping at some point. This doesn’t mean that during gameplay this is not possible but the graph can be left inconclusive at some cases (with increasing counters without a hard limit on the max value imposed by the e.g. entry demands).
The scene sets or the connections between them can be clicked to open the state explorer.
Press explore on some From/To instances and explore possible variable state conditions. When a matching state is found, “Configure debug from this” button is shown.
By clicking it, the debug window is opened.

2.8. Debugging

When the debug window is open, whatever state depiction is configured in it will be loaded when play mode is entered.

4. Programming considerations

3.1. State serialization and deserialization

GameGraph.GetStateAsJson() and ***GameGraph.SetStateFromJson(string data)*** can be used to save and load the state to/from string.
The system needs to be readied at this point, which can be checked with ***GameGraph.IsReady***.

3.2. Manually activating the system

If the system has been configured not to start automatically,
GameGraph.Instance.StartPrimedGraph() can be used to load the first scene set.

3.3. Extending the system

3.3.1. Representations

Deriving a class from ***Representation*** and overriding ***OnRepresentationActivated(bool skipToEnd)*** and ***OnRepresentationDisabled(bool skipToEnd)*** can be used to create new scene hooks for the system.
RepresentationEvents can be used as an example of doing this.

3.3.2. Modifications

All modifications derive from ***Modification*** and use the ***Apply*** method to change state data. The method takes in the current state and can access variables with ***state.AccessVariable(VariableContainer originalReference, out Variable var)***.
The originalReference is assigned from the editor and is used as an identifier where the out Variable is a reference to the runtime data container or whatever type the variable happens to be.
The builtin types are IntVariable and BoolVariable.
Use ***VariableContainerType(typeof(variableType))*** attribute to require a specific type of variable for the modification.
Additionally the modifications can change the scene set by using ***state.ChangeActiveSceneSet(SceneSet destination, LoadingScreen overrideLoadingScreenPrefab)***.

3.3.3. Modifiers

All modifiers derive from ***Modifier*** class. The most simple of these is ***ModifierOnDemand*** which demonstrates that ***Apply*** method triggers the modification itself.

3.3.4. Variables

Variables derive from ***Variable<datatype>*** and have to implement ***IsEqual(datatype other)***, ***IsLarger(datatype other)***, ***IsSmaller(datatype other)*** methods. Datatype denotes the contained type. The builtin types are bool and int.

3.3.5. Loading screens

Loading screens derive from ***LoadingScreen*** class. They should implement ***StartFadingIn()*** and ***StartFadingOut()***.
CanChange should be set to true when the loading screen covers the entire screen. This means that the system can behind the scenes load the new scenes in. After that is complete, the system will call ***StartFadingOut()*** which should have a mechanism to self-destruct after any possible animations are complete.
See DefaultLoadingScreen for an example.