Contents

RelationsInspector

The RelationsInspector is a Unity editor extension that lets the user visualize and edit relations between all kinds of project data. It will be called *RI* in this manual, to avoid confusion with Unity's inspector window.

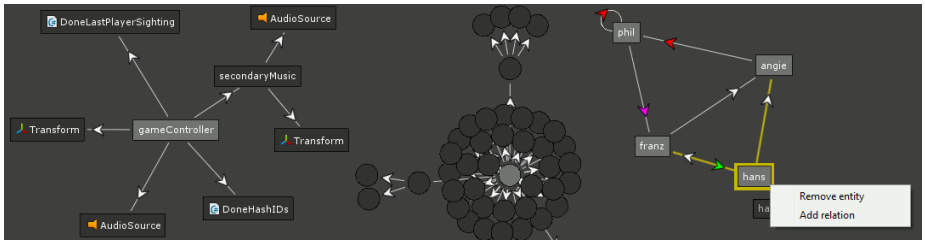

Figure 1: screenshots

Each kind of relation graph is driven by a backend class. A backend defines what relations to show for a specific object type.

Using the RI

Typically, you drag and drop some assets (the target objects) into the RI window. The RI will pick a backend that fits their types and show a graph made by the backend from your target objects. If the auto-selected backend is not the one you want, you'll find a dropdown in the toolbar where you can choose between

all backends that fit your target types. Alternatively, you can use the API to set target objects and pick the backend type from within your tool code.

The toolbar also contains these controls:

- buttons for navigating back and forth through the graphs you opened
- a **Clear** button, that will reset the window
- a **Rebuild** button, that will generate a new graph from the current target objects
- a **Relayout** button, that will re-run the layout algorithm on the current graph
- a **Layout** selector, that lets you arrange the graph in a tree layout, if applicable
- a **Entity widget** selector, that offers two ways of displaying entity nodes
- a menu that contains settings and usefull links

The rest of the window contains the graph and its minimap, as well as backend-specific GUI controls, if there are any. The graph area responds to the following inputs:

- dragging objects into the window makes them the new inspection targets. If *control* is held down at the same time, the objects are added to the existing inspection targets.
- moving the mouse wheel zooms in and out. When holding down *control* or *shift* the zoom will only affect the horizontal or vertical axis
- dragging the graph-area while the right mouse button is pressed, shifts the graph
- clicking into the minimap makes the window focus on the clicked location
- left-clicking an entity widget will select it. Pressing control at the same time adds the entity to the existing selection.
- dragging the graph-area while the left mouse button is pressed, selects all entities in the drag area
- right clicking on entity- and relation widgets may open a context menu, allowing the folding/expanding of subgraphs as well as backend-specific actions

Skin

RI ships with a light and a dark skin, which fit in with Unity's two skins. Each skin contains style settings for the entity widget, relations and the minimap. You can access the active skin through the gears menu (far right on the toolbar) or find them at `Assets/RelationsInspector/Editor/RelationsInspectorResources`. Since backends have full control over drawing the entity widget, they may choose to ignore the skin. By default, the selected nodes are marked by a blue or yellow outline, expandable nodes are marked by a red outline.

Settings

The following configuration options are available in the gears menu, and at `Assets/RelationsInspector/Editor/RelationsInspectorResources/Settings`
* **Cache layouts** if enabled, the layout of graphs you had opened before will be restored. Otherwise the default layout will be generated each time a graph is loaded.
* **Max graph nodes** At this threshold the graph generator will stop expanding nodes, to accomodate CPU and screen space limitations. The threshold is also applied to each manual node expansion. * **Tree root location** Determines in which direction the graph grows, when tree layout is used. * **Show minimap** Whether or not to display the minimap. * **Minimap location** Where to display the minimap. * **Log to console** If enabled, the window's log messages are forwared to Unity's logger. * **Invert zoom** Toggles the mousewheel zoom direction.

Automatic Backend

The RelationsInspector comes with a set of attributes that allow you to inspect objects without writing a backend class for their type. The attributes are:

- `AutoBackend`– indicates that RI should make an automatic backend for the marked type
- `Related`– indicates that the marked object(s) are related to this object
- `Relating`– indicates that the marked object(s) are relating to this object

Here is a simple example of an item upgrade type, where each object unlocks a number of dependent items.

```
using UnityEngine;
using System.Collections.Generic;
using RelationsInspector.Backend.AutoBackend;

[AutoBackend]
public class Upgrade : MonoBehaviour
{
    [Related]
    public List<Upgrade> unlockedUpgrade;
    public int attackStrength;
}
```

With that, you can now inspect objects of your type. The backend dropdown list will contain an entry named "RIAutoBackend of YOURTYPENAME".

Backend Development

A backend is a C# class that tells the RI what kind of relations it should display, how they should be displayed, and how the user should be allowed to modify them.

RI ships with a few example backends, but for your own classes and usecases, you may need to define new backends. By extending the default backend and with the help of utility classes, you can easily pick and choose which backend features you want to customize. If you don't want to implement a backend for your types, try an auto-generated one.

Getting started

First, you need to decide on the type of your entities, relations and how to define a relation in your graph. As an example, we'll develop a GameObject hierarchy backend. It will accept GameObjects dragged from the hierarchy window and show their child GameObjects. In this case it seems straighforward:

- entity type: GameObject
- relation definition: an entity B is related to another entity A, if A's transform contains B's transform
- relation type: can be anything. We only have a single kind of relation (parent->child). With no other kind to distinguish it from, any type will do. We pick string, arbitrarily.

With that, we can implement our backend's first version.

```
using UnityEngine;
using System.Collections.Generic;
using RelationsInspector;
using RelationsInspector.Backend;

public class TestBackend1 : MinimalBackend<GameObject, string>
{
    public override IEnumerable<Relation<GameObject, string>> GetRelations( GameObject entit
    {
        // parent -> entity
        if ( entity.transform.parent != null )
            yield return new Relation<GameObject, string>( entity.transform.parent.gameObjec

        // entity -> children
        foreach ( Transform t in entity.transform )
            yield return new Relation<GameObject, string>( entity, t.gameObject, string.Empt
    }
}
```

Save that to *Assets/Editor* or any other editor folder, wait for Unity to recompile, and now the RI backend dropdown should have a new entry called *TestBackend1*. Select it, drag scene objects into the window and you'll see their hierarchy tree.

We derive from *MinimalBackend* (one of the utility classes) instead of implementing *IGraphBackend* directly, because it contains default implementations of all interface members, and allows us to focus on the key properties: the entity- and relation type become generic arguments of the backend type, and the relation definition becomes the code of *GetRelations*.

In the following sections, we'll add more features to the backend.

Changing relations through context menus

Let's allow the user to modify the hierarchy by adding and removing relations. We'll do that with context menus for the entity and relation widgets. Add the following code:

```
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
using System.Linq;
using RelationsInspector;
using RelationsInspector.Backend;

...

    public override void OnEntityContextClick( IEnumerable<GameObject> entities, GenericMenu
    {
        menu.AddItem( new GUIContent( "Add as child" ), false, () => api.InitRelation( entit
    }

    public override void CreateRelation( GameObject source, GameObject target)
    {
        Undo.SetTransformParent( target.transform, source.transform, "Adding as child" );
        EditorUtility.SetDirty( source );
        EditorUtility.SetDirty( target );
        api.AddRelation( source, target, string.Empty );
    }

    public override void OnRelationContextClick( Relation<GameObject, string> relation, Gene
    {
        menu.AddItem( new GUIContent( "Un-child" ), false, () => DeleteRelation( relation.So
    }

    void DeleteRelation( GameObject source, GameObject target, string tag )
```

```
    {
        Undo.SetTransformParent( target.transform, null, "Un-childing" );
        EditorUtility.SetDirty( source );
        EditorUtility.SetDirty( target );
        api.RemoveRelation( source, target, tag );
    }
```

(Complete file)

Removing a relation is straightforward. Our deletion function disconnects the GameObject transform and tells the API to remove the relation edge from the graph.

Adding a relation requires one more step: the user has to select a second entity. That is handled by the *InitRelation* API call. *CreateRelation* is where we finally connect the transform and make the API add an edge to the graph.

Depending on your usecase, there could be constraints on how entities can be related. The scene hierarchy does not allow the child of a gameobject to also be it's parent, or gameobjects to be their own parents. You have to enforce all such constraints in *CreateRelation*. Without such restictions, RI will happily allow any of the following: objects being related to themselves, multiple relations between a pair of objects, cyclic relations, disjoint subgraphs, object without relations.

Adding a toolbar and GUI controls

The *OnGUI* backend method lets us draw GUI controls to the window, and expects us to return a Rect, describing the remaining space, which will be used for drawing the graph.

As an example control, we'll add a searchbar. We'll update the node selection based on the search string.

Add a variable `string searchstring;` to the class, also add this function:

```
    public override Rect OnGUI()
    {
        GUILayout.BeginHorizontal( EditorStyles.toolbar );
        {
            GUILayout.FlexibleSpace();
            searchstring = BackendUtil.DrawEntitySelectSearchField( searchstring, api );
        }
        GUILayout.EndHorizontal();

        return base.OnGUI();
    }
```

(Complete file)

Here we use the searchfield utility, which updates the entity selecting through the api for us, based on the value of *searchstring*. At the end *base.OnGUI* calculates the remaining window space for us.

Adding/Removing entites

You can add existing objects by dragging them into the RI window while holding down the CTRL key or passing them to the *AddTargets* or *AddEntity* API calls. The window toolbar is a good place for entity creation controls. Add this to your OnGUI method body:

```
if ( GUILayout.Button("Add GameObject", EditorStyles.toolbarButton, GUILayout.ExpandWidt
{
   api.AddEntity( new GameObject(), Vector2.zero );
}
```

Entity deletion is best done as a *OnEntityContextClick* handler:

```
menu.AddItem(new GUIContent("Delete entity"), false, () => { foreach (var e in entities) Del
```

Add this function:

```
void DeleteEntity( GameObject entity )
{
    api.RemoveEntity( entity );
    Undo.DestroyObjectImmediate( entity );
}
```

(Complete file)

Unity cleans up the broken transform references that result from removing a GameObject. For other entity types, you have to do this yourself: use the API's FindRelations method and clean them all up before removing the entity, like it's done here.

Node Widget UI

The Backend is responsible for drawing the node widgets, it gets a DrawContent call for each node on the screen. That call comes with the node's entity, position and a number of other parameters. It expects a Rect in return, which is used as the widget's bounding box when handling mouse events.

The default implementation of MinimalBackend looks like this:

```
public virtual Rect DrawContent( T entity, EntityDrawContext drawContext )
{
    return DrawUtil.DrawContent( GetContent( entity ), drawContext );
}
```

It maps the node's entity to a GUIContent and passes it to DrawContent, which will create a rect or circle widget from it. if you only need to customize the widget's icon, label or tooltip, simply overwrite GetContent:

```
public override GUIContent GetContent( T entity )
{
    return new GUIContent(YourLabel, YourIcon, YourTooltip);
}
```

If you want to draw anything other than GUIContent, you can draw your controls on top of the default rect and cicle containers by using the `DrawCircleAndOutline` and `DrawBoxAndBackground` utility methods. For example, this widget shows a slider control for the selected GameObject's layer value:

```
public override Rect DrawContent( GameObject entity, EntityDrawContext drawContext )
{
    // for unselected rects and all circle widgets, draw them the default way
    if ( drawContext.widgetType == EntityWidgetType.Circle || !drawContext.isSelected )
        return DrawUtil.DrawContent( GetContent( entity ), drawContext );

    // draw a slider in the rect widget

    // calculate the required area
    var sliderExtents = new Vector2( 155, 16 );
    var paddedSliderExtents = sliderExtents + 2 * new Vector2( 6, 6 );    // padding

    // draw background box
    var widgetRect = Util.CenterRect( drawContext.position, paddedSliderExtents );
    DrawUtil.DrawBoxAndBackground( widgetRect, drawContext );

    // draw the slider inside it
    EditorGUIUtility.labelWidth = 40;
    entity.layer = EditorGUI.IntSlider( Util.CenterRect( widgetRect.center, sliderExtent

    return widgetRect;
}
```

Layout caching

RI can restore the layout of graphs you had opened before. By default, it will do that only for backends that use Unity objects as entities, where it can rely on unique IDs. If you want to overwrite the default behaviour for your backend, for example because you expect the graph contents to change significantly between views, you can put the SaveLayout attribute on your backend class. `[SaveLayout( false )]` will disable layout saving, `[SaveLayout( true )]` will

enable it, even for entity types that are not Unity objects.

RelationsInspectorAPI reference

This API makes Relations inspector functionality accessible to your code. Backend classes get an API object by calling getAPI(1) in their `Awake` method, external tools can use the `GetAPI` method of `RelationsInspectorWindow`. In both cases the returned object is of type RelationsInspectorAPI.

```
void ResetTargets(object[] targets, bool delayed = true );
```

Clears the current graph and creates a new one for the given targets. If `delayed` is true, execution happens during the next update.

```
void AddTargets(object[] targets, bool delayed = true );
```

Clears the current graph and creates a new one for the union of existing and added targets. If `delayed` is true, execution happens during the next update.

```
object[] GetTargets()
```

Returns the current target objects.

```
void SetBackend(Type backendType, bool delayed = true );
```

Enforces selection of the given backend type. If `delayed` is true, execution happens during the next update.

```
void Repaint();
```

Draws a fresh view of the graph.

```
void Rebuild();
```

rebuild the graph from its current target objects

```
void Relayout();
```

redo the layout of the current graph

```
void SelectEntityNodes(System.Predicate<object> doSelect, bool delayed = true );
```

Makes the window select graph nodes according to the predicate. If `delayed` is true, execution happens during the next update.

Graph manipulation

```
void AddEntity(object entity, Vector2 position, bool delayed = true );
```

Adds the entity to the graph. The position is in graph coordinates, not window coordinates. If unsure, pass Vector2.zero. If `delayed` is true, execution happens during the next update.

```
void RemoveEntity(object entity, bool delayed = true );
```

Removes entity and all its relations from the graph. If `delayed` is true, execution happens during the next update.

```
void ExpandEntity(object entity, bool delayed = true );
```

Add relations of the given entity that have been hidden before, either due to the graph size limit or folding. If `delayed` is true, execution happens during the next update.

```
void FoldEntity(object entity, bool delayed = true );
```

Remove relations of the given entity. This will not affect relations that connect the entity to one of the target entities. If `delayed` is true, execution happens during the next update.

```
void AddRelation(object sourceEntity, object targetEntity, object tag, bool delayed = true )
```

Adds relation between the given entities to the graph. If `delayed` is true, execution happens during the next update.

```
void RemoveRelation(object sourceEntity, object targetEntity, object tag, bool delayed = tru
```

Removes the specified relation from the graph. If multiple matching relations exist, only the first one found will be removed. If `delayed` is true, execution happens during the next update.

```
void InitRelation(object[] sourceEntity, object tag, bool delayed = true );
```

Makes the UI initiate the creation of a new relation. The user then gets to pick the target entity, which will result in call to the backend's `CreateEntity`. If `delayed` is true, execution happens during the next update.

```
object[] FindRelations(object entity);
```

Returns all relations the entity is involved in.

IGraphBackend reference

To create a backend type, add a class to your project that implements this interface. Its two generic parameters are the graph entity type T and relation type P.

```
void Awake( GetAPI getAPI )
```

Called by the backend constructor. Calling getAPI(1) returns a RelationsInspectorAPI object which allows the backend to make graph manipulations, like adding/removing entities or relations, changing the inspection targets or the active backend.

```
IEnumerable<T> Init(object target);
```

This method generates graph entities from a target object, typically by just casting it. It is called after Awake, for each target object.

```
IEnumerable<Relation<T, P>> GetRelations(T entity);
```

Returns the relations which the given **entity** is a part of. It is used to grow the graph from the seed entities. Any relation can be returned by either its source or target entity or both.

Graph modification

```
void CreateRelation(T source, T target, P tag);
```

UI wants to create a new relation with the given properties.

Content drawing

```
Rect DrawContent(T entity, EntityDrawContext drawContext);
```

UI needs a rect, visually representing the given entity.

```
Color GetRelationColor(P relationTagValue);
```

UI needs the color in which to paint the relation marker.

```
string GetEntityTooltip(T entity);
```

UI needs a string to use as entity widget tooltip.

```
string GetTagTooltip(P tag);
```

UI needs a string to use as relation widget tooltip.

```
Rect OnGUI();
```

UI is being drawn. Backend gets a chance to draw its own controls. Returns the remaining space as rect. The graph is then drawn in that rect.

Other events

```
void OnEntityContextClick(IEnumerable<T> entities, GenericMenu contextMenu);
```

UI got a context click on the given entities. Items can be added to the given context menu.

```
void OnRelationContextClick(Relation<T,P> relation, GenericMenu contextMenu);
```

UI got a context click on the given relation's marker. Items can be added to the given context menu.

```
void OnEntitySelectionChange(T[] selection);
```

UI selection changed.

`void OnUnitySelectionChange();`

Unity selected objects changed.

`void OnDestroy();`

Backend object is about to be destroyed.

`void OnCommand(string command)`

A command event has been sent to the Relations inspector window. Use this to allow your own tool code to talk to your backend.