Animator Access Generator

Code generator utility for Unity game engine. Creates a class to conveniently access **Animator** states and parameters. The goal is to facilitate access to Animator states and parameters. Further on encapsulting access through dedicated methods makes it possible to detect potential problems and inconsistencies already at compile time.

Quick Start

Installation:

- Unity 4.3 is needed
- Download the current stable release **AnimatorAccess.zip** from GitHub
- Unzip and copy all files to some location under your Assets directory but **NOT under any editor directory**.

Generating Code:

- In the hierachy view select a game object that contains an Animator component having a valid controller
- Go to the new menu item **Animator Access** / **Create Animator Access**
- Choose a file name and the output directory where to place the C# file (has to be under Assets directory)
- Say *Yes* when the dialog appears about adding the component to the game object.

Usage example

Setup (related to provided ExampleScene.unity):

- Game object **ExamplePlayer** contains an Animator component
- Generated class is **ExamplePlayerAnimatorAccess.cs**
- Another component **Player** is attached to this game object too. It uses ExamplePlayerAnimatorAccess in its FixedUpdate method to control animations
- Animator states are:
  - **Idle**, **Jumping**, **Walking** and **Yawning** in layer 0 (**Base Layer**)
  - **Centered**, **Rotate-Left** and **Rotate-Right** in layer 1 (**Rot**)
- Animator parameters are:
  - **JumpTrigger** (trigger) and
  - **YawnTrigger** (trigger) and
  - **Rotate** (int) and
  - **Speed** (float)

To use **ExamplePlayerAnimatorAccess** define a member in **Player.cs** and assign a reference in Awake ():

```
AnimatorAccess.ExamplePlayerAnimatorAccess anim;
void Awake () {
    anim = GetComponent < AnimatorAccess.ExamplePlayerAnimatorAccess > ();
```

Now you have convenient access to all parameters and animator states. Aside from using parameter and state hash IDs directly, there are predefined methods for querying the state (prefix **Is**) and **Get** and **Set** methods to access parameters in a type safe way like **IsWalking () SetSpeed ()**:

```
void FixedUpdate () {
    currentState0 = animator.GetCurrentAnimatorStateInfo (0).nameHash;
    if (anim.IsWalking (currentState0)) {
        // set speed
        anim.SetSpeed (speed);
        // alternatively you can use hash IDs directly but this is more cumbersome:
        // animator.SetFloat (anim.paramIdSpeed, speed);
```

Events are another powerful way to use the generated component. The following code will register some event handlers which are called whenever a state or transition meets the given criteria:

```
void OnEnable () {
    anim.State (anim.stateIdIdle).OnActive += OnIdle;
    anim.TransitionTo (anim.stateIdWalking).OnStarted += OnStartedTransitionToWalking;
}
void OnIdle (AnimatorAccess.StateInfo info, AnimatorAccess.LayerStatus status) {
    // Called repeatedly on each cycle as long as state 'Idle' is active
}
void OnStartedTransitionToWalking (AnimatorAccess.TransitionInfo info, AnimatorAccess.LayerStatus status) {
    //  Called once every time a transition to state 'Walking' is starting
}
```

Concept And Workflow

Basic Idea

Whenever you have made any changes in the Animator window like adding, renaming or deleting states or parameters, you should update the animator access component. Animator Access Generator analyses the existing version of the component to be generated and works with a **two-step** procedure to handle changes:

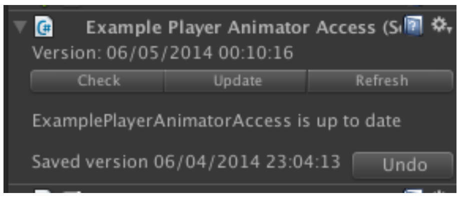
- Previously valid members (i.e. Animator parameters and states) are detected and marked with the *Obsolete* attribute
- Those members in the previous version that were already marked as *Obsolete* will be removed

The basic idea is to give you the chance to refactor your code without having uncompileable code. If there are any references to members that are not valid any longer, obsolete warnings guide you where you have to make changes:

(*CS0168: Animator state or parameter is no longer valid and will be removed in the next code generation…*)

Workflow Using The Custom Editor

The generated component has a custom inspector window:

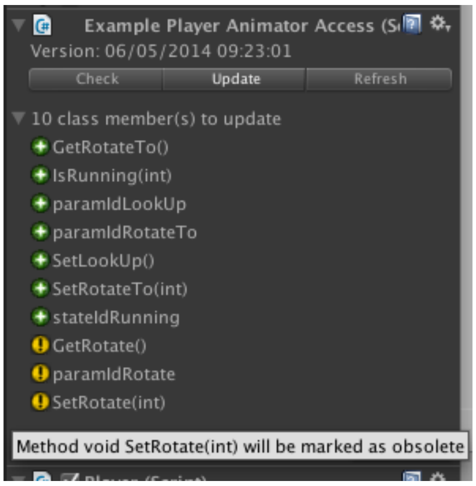

Example Player Animator Access (S
Version: 06/05/2014 00:10:16

ExamplePlayerAnimatorAccess is up to date
Saved version 06/04/2014 23:04:13

The status is updated every 30 seconds automatically. Use the *Check* button to force a status update immediately. *Check* will not perform any changes but is meant to preview what will happen on an update. Hovering the mouse over an entry will show a tooltip with more information

Supposed we made 3 changes: A new state **Running**, New trigger parameter **LookUp** and renaming of parameter **Rotate** to **RotateTo**

This will look like:


Example Player Animator Access (S
Version: 06/05/2014 09:23:01

10 class member(s) to update
GetRotateTo()
IsRunning(int)
paramIdLookUp
paramIdRotateTo
SetLookUp()
SetRotateTo(int)
stateIdRunning
GetRotate()
paramIdRotate
SetRotate(int)
Method void SetRotate(int) will be marked as obsolete

1. New field **stateIdRunning** and an **IsRunning ()** method for **Running**
2. Field **paramIdLookUp** and **SetLookup ()** method for **LookUp** (triggers don't have Get methods)
3. Field and methods for **RotateTo** including a Get method as this is an int
4. The code of the previously named parameter **Rotate** will be marked as obsolete i.e. **paramIdRotate**, **GetRotate** and **SetRotate**.

Push the *Update* button to perform the changes and regenerate the component's source code. Note that the changes are not refreshed automatically. Most often you will like to first view and adopt the changes in MonoDevelop and then run your game in Unity.

The *Refresh* button forces Unity to reload the changes from disk (s. *Advance Topics* on automatic refreshing).

Undo performs a roll back to the previous version. Note that there is only one backup saved. Again there is no automatic refresh so push *Refresh* after Undo.

Using The Generated Code in MonoDevelop

In this section we assume the example setup described in the **usage example above**. The code including a working example scene can be found in the provided Example folder s. **Scripts**. See section **Naming Conventions** below for more about how member names are built.

Accessing Parameters

For every Animator parameter a public integer field is generated that contains the parameter name hash. Depending on the parameter type, **additional Get- and Set-methods** are generated . For example an integer parameter *rotate* will produce the following 3 members in the generated class:

```
public int paramIdJumpTrigger;
public void SetRotate (int newValue) { ... }
public int GetRotate () { ... }
```

In general it is not recommended to use the paramIdJumpTrigger directly. Instead call the corresponding Get and Set methods. This makes your code more robust because the compiler will detect type mismatches already if you for example try to set an integer to a boolean parameter. So in your *Player.cs* you just write

```
anim.SetRotate (myValue);
```

Note that there will be no *Get* method for *trigger* parameters. On the other hand a float produces 2 *Set* methods like in class Animator.

Working with States

Like with parameters integer member fields are created for holding the name hash of each state. For checking the current state of a specific layer a bool method is generated:

```
public int stateIdIdle;
public bool IsIdle (int nameHash) { ... }
```

At the moment (release 0.8) the name hash of the layer has to be provided as argument. Maybe a future release will provide an overlodaded version without parameter doing the look up internally.

Events on States And Transitions

All generated AnimatorAccess components provide an interface to register listener callbacks for a wide range of use cases. Events occur when a specific condition is met that is related to states or transitions.

The condition is defined when a listener subscribes to a certain event by using **+=** operator as the standard **C# event mechanism** is used. That means there will be no notification as long as no listener has subscribed. Compared to Unity's SendMessage approach this design has **advantages regarding performance** as otherwise there would have been hundreds of SendMessage calls every second even if no component is interested.

Some examples for registering listeners:

```
anim.AnyState (0).OnChange += OnStateChangeLayer0:
    // called whenever the state changes in layer 0; Leave layer empty to observe all layers
anim.State (anim.stateIdIdle).OnEnter += OnIdleEntered;
    // called once when 'Idle' state is entered
anim.Transition (anim.stateIdIdle, anim.stateIdWalking).OnStarted += OnIdleToWalking;
    // called when the transition from 'Idle' to 'Walking' has started
anim.TransitionFrom (anim.stateIdJumping).OnActive += OnTransitionFromJumping;
    // called every frame as long as any transition from State 'Jumping' is ongoing
```

and their method definitions:

```
void OnIdleEntered (StateInfo info, LayerStatus status) { ... }
void OnTransitionFromJumping (TransitionInfo info, LayerStatus status) { ... }
```

Hint: If you feel that entering a state is too late for the action to perform, use a transition **OnStarted** event instead to *win* the transition time.

Performance: Registering a listener implies an entry in dictionary and a call to its check method every frame. As there are only a few lines of basic C# code executed, it should not have any perfomance penalties.

Runtime Details about States And Transitions

An AnimatorAccess component provides access to a couple of information about states and transitions that is otherwise not available at runtime. This lets you easily **look up the name**, speed, … for debugging purposes. The properties

```
public Dictionary<int, StateInfo> StateInfos;
public Dictionary<int, TransitionInfo> TransitionInfos;
```

provide dictionaries having the state / transition name hash as key:

- **StateInfo** contains clear text name, layer, layer name, tag, speed, …
- **TransitionInfo** contains clear text name, layer, source, destination, duration,

Transition name hashes are not provided an integer member variable like for states and parameters. Either you have an ID in case of an event listener and thus the TransitionInfo or you can iterate the TransitionInfos dictionary and search for whatever you are looking for. **Never rely on state or transition names!** Use them for debugging only.

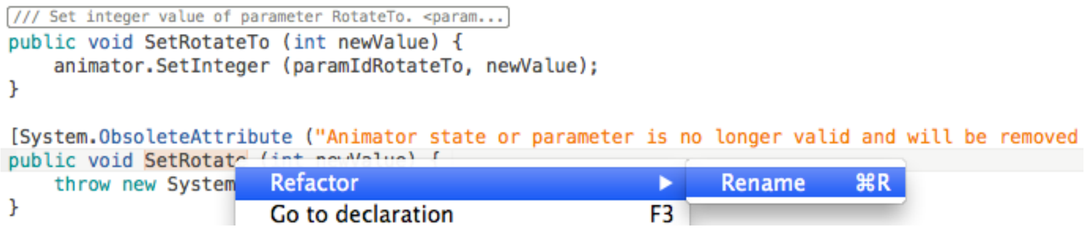
Both properties are initialised deferred at the time of the first access or when an event listener is registered. Thus if you are afraid of performance drawbacks don't use them and (refrain from events).

Renaming And Efficient Handling of Obsolete Warnings

Although generated code should not be edited, it can be pretty useful to do so temporarily to quickly update your other code. If you rename Animator parameters, states or layers that are referenced in your own code, you will get warnings about using obsolete members. While single occurrences are easy to maintain, renaming of a widely used parameter would be painful to replace in code.

To do this more efficiently use refactoring and rename the **obsolete** member to the new name. Suppose there are several calls to **SetRotate** in the example above. We want to change all these calls at once to point to the new method **SetRotateTo**:

Go to the **obsolete** method **SetRotate (int newValue)** in the newly generated file **ExamplePlayerAnimatorAccess.cs** and rename it to **SetRotateTo**.



```
/// Set integer value of parameter RotateTo. <param...
public void SetRotateTo (int newValue) {
    animator.SetInteger (paramIdRotateTo, newValue);
}

[System.ObsoleteAttribute ("Animator state or parameter is no longer valid and will be removed
public void SetRotate
    throw new System
}
```

Yes, there is already our new method SetRotateTo having exactly the same signature and now we have two of them. But that's the clue: After renaming is done, simple delete the **obsolete** version - ready. Now all references point to the correct method.

Errors in MonoDevelop

There are three known situations when you may get errors:

1. Obsolete warnings were ignored and the next *Update* was triggered. Then all obsolete members are removed and references to these will fail
2. You changed state and parameter hash prefix settings to contain the same string, removed a parameter and create a state with the same name
3. Generated code was edited so that a naming conflict arose.

If *Undo* does not help, most problems can be solved pretty similar the way we handled warnings. Do single changes manually and a bunch of changes with the refactoring strategy. If there are many places referring to a missing member, manually introduce it as dummy and then use refactoring.

Naming Convention And Customising

You can customise the way how to generate parameter and method names in the *Settings* window. All access methods start with **Is**, **Get** or **Set**. This cannot be changed but the pattern for the appended item name can be modified. You can for example define that all animator state query methods (Is…) contain the prefix *State* before the name e.g. IsStateIdle ().

Be careful with changes for the hash ID member variables prefixes. States and parameters should have **different prefixes** to avoid naming conflicts.

By default the **Layer name** is ignored for layer 0 and prepended for all other layers.

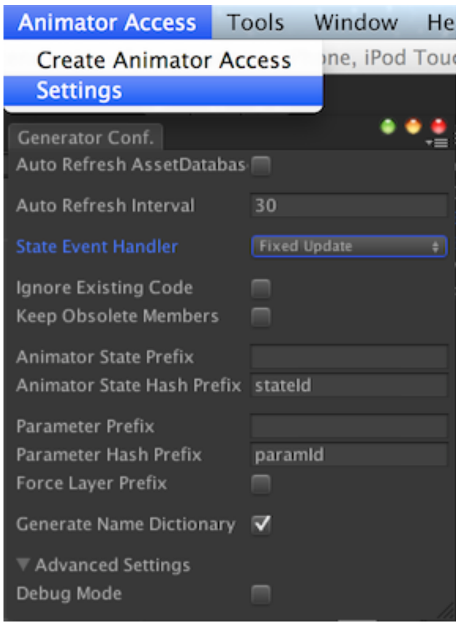
Avoid non-ANSI characters. Field names are converted to match the pattern [a-z_][a-zA-Z0-9_]. Non-matching characters are changed to an underscore.

Look at the tooltips in the *Settings* windows or go to Advanced section below for a complete description.

Advanced Topics

Settings

Configuration is done via the Settings window.



Auto Refresh AssetDatabas ☐
Auto Refresh Interval 30
State Event Handler Fixed Update
Ignore Existing Code ☐
Keep Obsolete Members ☐
Animator State Prefix
Animator State Hash Prefix stateId
Parameter Prefix
Parameter Hash Prefix paramId
Force Layer Prefix ☐
Generate Name Dictionary ✓
Advanced Settings
Debug Mode ☐

1. **Auto Refresh AssetDatabase:** Automatically call an AssetDabase.Refresh () after updating an existing AnimatorAccess class.
   Note that the MonoDevelop project is reloaded too which can be annoying.
2. **Auto Refresh Interval:** Automatically check for updates after this interval (in seconds) has elapsed. Set this to 0 to suppress automatic checking.
3. **State Event Handler** Select if and where to generate the code for automatic event handling and invoking of callbacks on state changes:
   FixedUpdate: Check for state changes and transitions is performed in FixedUpdate
   Update: Same for Update method
   None: you have to manually invoke method CheckForAnimatorStateChanges () of the generated class.
4. **Ignore Existing Code:** Unchecked (default) means that the current version of the class is analysed first. Existing members that are not valid any longer are created once but with the obsolete attribute set.
   So you can replace references to these outdated members in your code. Performing another generation will then remove all obsolete members.
   Check this if existing members should be removed immediately.
   Note that this option overwrites 'Keep Obsolete Members'
5. **Keep Obsolete Members:** Obsolete fields and methods are not removed. This might lead to a lot of unnecessary code.
   Note that this option is not considered if 'Ignore Existing Code' is set.
6. **Force Layer Prefix:** Check this if you want the layer name be prepended even for Animator states of layer 0.
7. **Animator State Prefix:** Optional prefix for all methods that check animation state. Prefix is set betwen 'Is' and the state name.
   Example:
   'State' will generate the method 'bool IsStateIdle ()' for state Idle.
8. **Animator State Hash Prefix:** Optional prefix for all int fields representing an animator state.
   Example:
   'stateHash' will generate the field 'int stateHashIdle' for state 'Idle'.
9. **Parameter Prefix:** Optional prefix for parameter access methods. Prefix is set betwen 'Get'/'Set' and the parameter name.
   Example:
   'Param' will generate the method 'float SetParamSpeed ()' for parameter 'Speed'.
10. **Parameter Hash Prefix:** Optional prefix for int fields representing a parameter.
    Example:
    'paramHash' will generate the field 'float paramHashSpeed' for parameter 'Speed'.

11. **Debug Mode:** Extended logging to console view.

Persistent Storage Location

All settings are saved in directory at:

```
Application.persistentDataPath + "/AnimatorAccessConfig.txt":
```

See the **Unity documentation** or use Debug.Log to find the OS specific place.

AnimatorAccessConfig.txt is a simple text file. The modified PlayerPrefs class is based on PreviewLabs.PlayerPrefs (s. **their web site** for original source and description).

The *Undo Files* are stored using Unity's **Application.temporaryCachePath**.

SmartFormat Template

SmartFormat is currently used as template enginge to generate the code. There are OS dependent template files due to different line endings on UNIX and Windows. The templates can be found under: *InstallationDir*/Editor/Templates/DefaultTemplate-UNIX.txt *InstallationDir*/Editor/Templates/DefaultTemplate-Win.txt

Moving Animator Access Menu

If you want to move the menu for Animator Access from top menu to a different location:

Edit the constant **RootMenu** in file **Editor/MenuLocation.cs**.

Note that you have to **repeat this step after every update of Animator Access**.

Git Subtree Integration / Contributing

If you are using Git in your project, a *git subtree* for AnimatorAccess might be the easiest way to integrate it. Especially when you plan changes or like to contribute to AnimatorAccess there are some conventions about the **.meta files**. Basically .meta files should not be commited except for the *Example* folder. To integrate this smoothly the following best practice is recommended:

Given a subtree in Assets/ThirdParty/AnimatorAccess set up the following .gitignore structure:

- Assets/ThirdParty/.gitignore contains
  AnimatorAccess/.gitignore
- Assets/ThirdParty/AnimatorAccess/.gitignore contains
  **.meta*
  Example/.gitignore
- Assets/ThirdParty/AnimatorAccess/Example/.gitignore contains
  !.meta*

This ensures that only .meta files under Example are committed and .gitignore files are held locally

File Specific Configuration

This is currently not supported via GUI but can be done easily in code. The responsible Config factory class is designed as partial class so that it can be extended in a separate file. In order to register your own **SpecialConfig** create a new file ConfigFactoryExt.cs and implement **ConfigFactory**'s static constructor.

```
namespace Scio.AnimatorAccessGenerator
{
    public partial class ConfigFactory {
        static ConfigFactory () {
            ConfigFactory myFactory = new ConfigFactory ();
            instance = myFactory;
            myFactory.defaultConfig = new Config ();
            Config specialConfig = new SpecialConfig ();
            specialConfig.AnimatorStatePrefix = "MyAnim";
            myFactory.configs ["ExamplePlayerAnimatorAccess"] = specialConfig;
        }
    }
    // Provide an implementation of your **SpecialConfig** class. Bear in mind that the default class makes all
    // changes persistent.
    public class SpecialConfig : Config {
        public override string AnimatorStatePrefix {
            get {
                return "MyAnim";
            }
            // provide an empty setter to avoid that "MyAnim" is written back to the default config.
            set {}
        }
    }
}
```

Extending BaseAnimatorAccess

BaseAnimatorAccess extensions can be done the same way like *ConfigFactory*. The class is declared as *partial* and thus can be extended by own interface methods. Further on you need to create a handler that is called:

```
namespace AnimatorAccess
{
    public partial class BaseAnimatorAccess
    {
        public AnyStateOn2LayersHandler AnyStateOn2Layers (int layer1, int layer2)
        {
            AnyStateOn2LayersHandler handler = new AnyStateOn2LayersHandler (layer1, layer2);
            int id = handler.GetHashCode ();
            if (!StateHandlers.ContainsKey (id)) {
                StateHandlers [id] = handler;
            }
            return (AnyStateHandler)StateHandlers [id];
        }
    }

    public class AnyStateOn2LayersHandler : AbstractStateHandler
    {
        public AnyStateOn2LayersHandler ((int layer1, int layer2) {}

        public override void Perform (LayerStatus[] statuses, Dictionary<int, StateInfo> stateInfos) {}
    }
}
```