1. Source Code Home . . . 2
1.1 Introduction . . . 2
1.2 Setup . . . 2
1.3 Building Source Code . . . 4
1.3.1 Deprecated Build Environments . . . 5
1.3.2 Sharing Your Built Editor with Your Team . . . 6
1.4 Code Conventions . . . 6
1.4.1 Naming Conventions . . . 6
1.4.2 Code Formatting . . . 8
1.4.3 Design Considerations . . . 17
1.5 Tutorials . . . 18
1.5.1 Create a new Component to Unity in C++ . . . 18
1.5.1.1 Part 1: Create a New Behaviour . . . 18
1.5.1.2 Part 2: RTTI (run-time type information) . . . 20
1.5.1.3 Part 3: Script Binding and JAM . . . 21
1.5.1.4 Part 4: Expose to Editor . . . 22
1.5.1.5 Part 5: Custom Inspector . . . 23
1.5.1.6 Part 6: Bind to gameObject . . . 23
1.6 Adding Script bindings to a C++ class . . . 23
1.7 Allocating memory . . . 25
1.8 Managing and Building Resources . . . 28
1.8.1 Built-in Resources . . . 28
1.8.2 Editor Resources . . . 29
1.9 Multithreading in Unity . . . 30
1.10 Blobification . . . 34
1.11 Multithreaded Rendering . . . 37
1.12 Editor . . . 38
1.12.1 How to serialize stuff (C#) . . . 38
1.12.2 Style Guide . . . 39
1.12.3 Working with Editor Styles . . . 40
1.12.4 Writing Custom Inspectors . . . 43
1.12.5 Writing GUI Controls . . . 44
1.13 Tips&Tricks . . . 48

Source Code Home

This is the home of the Source Code space, where we document how to work with the Unity source code.

> Content in the **Source Code** space is used both by **Unity Employees** and by **Source Code Customers**.

Hello source code user! Welcome to your informational hub for instruction & advice on working with the Unity source code and using it in the most awesome possible ways.

If you're new to the source or have been away for a few versions, make sure to check out the Setup and Building Source Code instructio ns. Otherwise, feel free to browse around in order to learn many interesting things.

Have fun,

Your friends at Unity Technologies

General Topics

Setting up your computer to build Unity source code for all supported target platforms
Building Unity source code
Debugging the Unity Editor and your built runtime applications
Modifying and extending Unity source code to customize the engine & tools to your needs

Introduction

Hello source code user! Welcome to your informational hub for instruction & advice on working with the Unity source code and using it in the most awesome possible ways.

If you're new to the source or have been away for a few versions, make sure to check out the Setup and Building Source Code instructions. Otherwise, feel free to browse around in order to learn many interesting things.

Have fun,

Your friends at Unity Technologies

General Topics

Setting up your computer to build Unity source code for all supported target platforms
Building Unity source code
Debugging the Unity Editor and your built runtime applications
Modifying and extending Unity source code to customize the engine & tools to your needs

Setup

Setting Up Your Machine

The Basics

Windows

Operating System

The minimum supported version of Windows for general source code development is Windows 7 SP1. If you are building or using the Windows 8, RT, or Windows Phone 8 runtime engines, you must have Windows 8.

Tools

For Windows development, you'll need to install the following tools (at minimum):

- **Visual Studio 2010**
  - The Visual Studio 2010 Service Pack 1 can be downloaded from Windows Update, if you have enabled the "Get updates for

other Microsoft products" option.
- **32-bit Perl interpreter** - You can download the interpreter directly from ActivePerl at http://www.activestate.com/activeperl/downloads.
- Note that to build successfully, your Unity source must be located in a path without spaces on a HFS+ (Mac) file system.

Mac OS X

Operating System

You can participate in Unity development using OS X Snow Leopard (10.6), OS X Lion (10.7), or OS X Mountain Lion (10.8).

Tools

For OS X development, you'll need to install the following tools (at minimum):

- **Xcode**- Install the latest Xcode from the Mac Store.
  - In newer Xcode versions which come from the Mac App Store (ie without an installer), you need to manually install the command line tools for Xcode (in Preferences -> Downloads).
  - Make sure Xcode.app is located directly in the Applications folder - don't move it to e.g. a sub-folder, and make sure it's named "Xcode.app."
  - If you're building 3.x source code:
    - Xcode 4.3 and newer will no longer install into the /Developer folder, but Unity's build scripts still expect the SDKs to be located there. Until that is fixed, you can create a symlink to point to the right SDK location: `sudo ln -s /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/ /Developer`
    - Xcode 4.4 and newer will no longer include the 10.6 SDK required to build Unity. Get it from *smb://homes/Software/Mac/Developer Tools/MacOSX10.6.sdk.zip* and unzip to `/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/`.
      - Link libstdc++.dylib -> libstdc++.6.dylib in `/Developer/SDKs/MacOSX10.6.sdk`
  - If you're installing Xcode 3.6.* on Mountain Lion, you need to disable System Development Tools from the installer menu. The CHUD tools are only shipping in 32 bit, and OSX 10.8 is 64-bit only for kernel extensions
    - If you didn't know this when you installed and are now getting kernel panics on boot, boot into target disk mode and delete CHUDKernLib.kext, CHUDProf.kext, CHUDUtils.kext, AppleProfileFamily.kext

Linux

Operating System

You can participate using Ubuntu 12.04 or later.

Setting Up a Development Installation

- Download and install Ubuntu (or your distro of choice, but these instructions are for Ubuntu)
  - Choose 64-bit. We're ninjas, we don't use "recommended" options :-)
  - In the **Preparing to Install Ubuntu** step, make sure to check:
    - Download updates while installing
    - Install this third-party software (these are mainly drivers)
  - After installation completes, and you've restarted your system:
    - You'll get a notification that "Restricted drivers" are available - install/enable the relevant ones
      - Do **NOT** use the "post-release updates" versions
    - Launch "Update Manager" and install the available updates
- Install required packages:
  - `$ sudo apt-get install mercurial eclipse eclipse-cdt monodevelop build-essential g++-multilib autogen autoconf libtool nasm libglu1-mesa-dev libtool flex bison libxrandr-dev libxrender-dev libxcursor-dev ia32-libs p7zip-full libgtk2.0-dev`
  - If you're on 64-bit OS, and also want to cross-compile 32bit targets:
    - `$ sudo apt-get install gcc-multilib g++-multilib libx11-dev:i386`
- `perl build.pl -target=Linux64StandalonePlayer`
- That should be it!

Gotchas:

- Have problems linking against `libstdc++`?
  - See if you have `/usr/lib32/libstdc++.so`. If you don't, try this: `sudo ln -s libstdc++.so.6 /usr/lib32/libstdc++.so`
- As of mid-june 2012, our SSE code kills gcc-4.6 so probably have to use gcc-4.5. Fixed upstream but not in main repos yet.
  - `$ sudo ln -sf gcc-4.5 /usr/bin/gcc`
  - `$ sudo ln -sf g++-4.5 /usr/bin/g++`
- If you're cross-compiling 32-bit stuff on a 64-bit OS, you'll probably need to add some symlinks after installing `:i386` libs:

- `$ sudo ln -s libXrandr.so.2 /usr/lib/i386-linux-gnu/libXrandr.so`
- `$ sudo ln -s libXrender.so.1 /usr/lib/i386-linux-gnu/libXrender.so`
- `$ sudo ln -s libGLU.so.1 /usr/lib/i386-linux-gnu/libGLU.so`
- `$ sudo ln -s libXext.so.6 /usr/lib/i386-linux-gnu/libXext.so`
- `$ sudo ln -s libXcursor.so.1 /usr/lib/i386-linux-gnu/libXcursor.so`
- `$ sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/mesa/libGL.so`
- `$ sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so /usr/lib/i386-linux-gnu/libGL.so`

- Don't want a popup dialog every time something crashes? Remove the apport package.

Building Source Code

This page details the entire workflow needed to build Unity source code.

- Building
  - Build Entry Point
  - Build System Layout
  - Build options for the development environment
  - Running build.pl
  - Building using Xcode or Visual studio
  - Troubleshooting

Building

Build Entry Point

The Perl script called *build.pl* is single entry point to the system. *build.pl* resides in the root of the clone. TeamCity builds targets by invoking build.pl directly; the solution files for Visual Studio and project files for Xcode invoke the Jam buildsystem directly.

Build System Layout

Important locations in the buildsystem:

- build.pl
  - Placed in the root of the checkout, an entry point to the build system on both windows and mac.
- Tools/Build
  - Location to place build related modules. There is a *Tools.pm* and a *MacTools.pm*
- Tools/Build/TargetBuildRecepies
  - Location of target modules. Each modules here should have one or more target functions capable of building specific targets. They should contain the information needed to register build targets (like build dependencies, possible options).

Build options for the development environment

There are 3 kinds of build options you can set up with the ***build.pl*** script. Build machines have certain release defaults and developers (executing build.pl from command line) sometimes have other defaults.

- *-UnityDeveloperBuild=0/1*
  - This affects how unity configure header files are generated, which sets up various defines. Like if autoupdating should be performed in the webplugin. A developer usually want's this to be enabled, all release versions have this disabled. By default, when you build a target with the build.pl script, this option is disabled, **-prepare** and **-setupWebPlayer** enable it by default.
- *-codegen=release/debug*
  - Build a release or debug build. The build.pl script by default sets this to debug, release builds are done by build machines. The main difference it compiler optimizations and debug symbols (on windows they are always included).
- *-developmentPlayer=0/1*
  - This enables or disables the profiler and reproduction log defines. They are used by the regression frameworks and profiler enabled builds in standalone and web players.

 **Warning:** The '-developmentplayer=0' and '-developmentplayer=1' options output their build targets to the same directory. If you build the non-development player for a platform and then build the development player, the development player will overwrite the non-development version. See Building the Editor for Internal Distribution for details on how to create a build of the Unity Editor that you can distribute internally with the playback engines in their correct locations.

Other important options include:

- *--prepare / --scaffold*
  - Only perform scaffolding of the necessary files for all targets, so the targets will work when built by xcode/VS. When a *-targe t* list is provided with the prepare command only those targets are affected.

- *--reset*
  - Same as the above command except that it completely cleans the build folder of old/stale files. It's good to run this once in a while for proper cleanup. A fatal error occurs if any file/directory fails to be removed.
- *--workspace*
  - Unpacks the minimum required set of builds.zip/7z files and generate/update Unity Visual Studio project. Much faster than -prepare
- *--processTxts*
  - Process the txt files. Cpp files will be generated as well as the UnityEngine and UnityEditor dlls. This is the same as running *Tools/cspreprocess -nodocs*.
- *--setupWebPlayer*
  - This sets up the web player files in the build directory and links them with the install location on the machine (symlinks on mac, junctions on win). Then the web player is debuggable straight right from xcode/VS. The web plugin must also be built and exist in the build directory.

Running build.pl

If you execute build.pl with no arguments, you will be provided with a console menu where you can select which targets to build (by entering a number followed by <enter>). If you know the name of the target you want to build, you can build it as an argument to build.pl:

```
$ perl build.pl --target=WindowsEditor
```

Multiple targets can be combined with one run:

```
$ perl build.pl --target=WindowsEditor,WindowsStandalonePlayer
```

Building using Xcode or Visual studio

If you execute build.pl with the *--workspace* option, or select *prepare* from the build.pl menu, it will generate Xcode and VS projects for you. These are located in:

- Windows: open Projects/JamGenerated/_workspace_.vs2010/AllTargets.sln
- OS X: open Projects/JamGenerated/_workspace_.xcode/AllTargets.xcodeproj

These will still use the jam system for building internally, but should allow you to navigate, build and debug your project as you are used to from your IDE.

Troubleshooting

If you get a linker error like this:

```
LINK : fatal error LNK1123: failure during conversion to COFF: file invalid or corrupt
```

Reinstall Visua Studio 2010 SP1 from http://www.microsoft.com/en-us/download/details.aspx?id=23691.

Deprecated Build Environments

Deprecated Windows development environment (for 4.0< branches)

- Build dependencies:
  - ActivePerl 5.10
  - Subversion command line client 1.6.6
  - Visual Studio 2008 can be downloaded from here (only with at the office of through VPN)
    - To skip needing to burn the VS iso to CD it's possible to mount the iso with this tool, The guide to using the tool is in a text file within the zip. You can also use Virtual Clone Drive (easier to setup).
  - Visual Studio 2008 Sp1 can be downloaded from here

Important changes in 3.x from the 2.x build system

- **There is only a single entry point to the system, which is *build.pl* in the root of the checkou**t. No helper scripts need to or should be run seperately or from the build scripts. All build process functionality goes into the *Tools/Build/** modules. Some artifacts from the old system haven't been fully converted to it so this rule isn't absolute at the moment.
- One important change for the Mac is that we **no longer symlink all the frameworks** (this used to be done extensively in installframeworkshack.command). Copies should be used but hard links are used since they are faster. There are a few reasons for this change: some files will probably be unique in the future but are used by all right now (like Mono stuff being different for webplayer,standalones,editor). Symlinking was error prone when setting up the new system, it is dependent on relative paths staying the same and won't complain about symlinking to non-existent files. We want the build system to be very rigid and complain about any error since otherwise they might go unnoticed (until too late). Making the mac and windows platforms work/build in similar ways could be advantageous to people who work on both platforms, expecially those who are not used to working in the other one. However, this doesn't need to be an all out change, if there are arguments for using symlinks instead, like if they make develpment easier, then this can be changed for those cases, preferably by making a wrapper function for symlinking files which does error checking and isn't as dependent on relative paths staying the same.

Deprecated Windows development environment (for 2.6.1< branches)

- Download the required software package for 7zip, ActivePerl, NSIS and an SVN command line client. This also contains a perl module dependency so you don't need to install it by hand (if running build machine scripts).
    - Some packages there are out of date! Better install all packages manually
    - Run `InstallAll.bat` to install everything, just need to click through the dialogs.
    - When setting up an automated build machine:
        - Run `AddBuildMachineUser.pl` to automatically create a buildprocess user for the file share.
        - Run `SetupBuildMachineFolders.pl` to create the needed dirs, copy script files and enable file sharing with appropriate permissions. It's probably neccessary to edit the file a little bit to set the base path unless the default is ok (C:\builds\unity-runk and checkout in C:\builds\unity-trunk\unity) (see the source building guide)
        - Run `SetupCodeSigningCertificates.bat`. This is script requires user intervention to click through certificate import dialogs.
- [optional] Also download the optional software package which contains TortoiseSVN, Notepad++ and WinSCP which are just useful tools for windows development.
- If you are skipping the copy protection, make sure to disable it in `Configuration/BuildConfig.pm` and disable recursive revert as well in script/svn_update.pl (line 40)
- When building the example project it is necessary to activate Unity (like when doing this on a build machine), or else building will fail after 30 days.

Sharing Your Built Editor with Your Team

Code Conventions

Internal Coding Conventions

The following pages document our coding conventions for all internal code, meaning code that isn't seen by our users. Except where otherwise noted, the following conventions apply to both C++ and C# code.

Both C++ and C#

- Naming Conventions
- Code Formatting
- Design Considerations

C++

- C++ File Layout
- C++ Header Example
- C++ Source Example

C#

- TODO

Public Coding Conventions (docs, demos, tutorials)

The following page document coding conventions for all public code, meaning code that is seen by our users. This includes scripts in example projects and tutorials, and example code snippets in the scripting reference. Except where otherwise noted, the conventions apply to both JavaScript and C#.

- Public Code Conventions

Naming Conventions

General

- Spell words using correct English spelling.
    - Use all upper case for abbreviations (`ID`, `CPU`, `XML`, etc.) unless the conventions mandate this part of the name should start with a lower case letter.
    - Avoid abbreviations when possible unless the abbreviation is commonly accepted.
    - Using commonly accepted names such as `i` and `j` are allowed for local iterator variables.

```
class XMLDocument
{
    ...
};

// local variable
XMLDocument xmlDocument;
```

- Use descriptive and accurate names, make them longer if necessary.
- Do not use names containing double underscores __ or names starting with an underscore followed by an uppercase letter.
  Rationale: These names are reserved for use by the compiler and standard library.

File names (C++, C#)

- Use upper CamelCase for file names.
- C++ implementation files end in `.cpp`
- C# files end in `.cs`
- Name the files after the main class they define.

File names (TXT)

- Name the files after the main class they define with Bindings suffixed. (e.g. WebcamTextureBindings.txt)
- TXT files should contain a single class with the exception of classes/enums that are tightly coupled to it. (e.g. SerializedObject and SerializedProperty)
- **Do not start moving classes out of the bloated txt files yet. There are plans to do this in the future and we need to do it in a unified effort.**

Class and struct names (C++, C#)

- Do not add a prefix to classes, template classes, interfaces or structs.
- Use upper CamelCase for names.

Functions (C++, C#)

- Use upper CamelCase for names.
- Arguments use lower camelCase.

```
int MyNewFunction (int anIntegerArgument);
```

Variable names (C++, C#)

- Never use Systems Hungarian notation.
- Occasional Apps Hungarian notation is allowed.
- Local variables use lower camelCase.

```
int thisIsALocalVariable;
```

- Member variables are prefixed with `m_`, rest is upper CamelCase.
- This applies to both private and internal variables. Note that at the moment we never expose member variables in our public API; only properties, so the user will see the lower camelCase property, not the `m_` prefixed member variable.
- C++ Only: All public member variables also use the `m_` prefix with the rest in upper CamelCase.

```
int m_ThisIsAMemberVariable;
```

    - **EXCEPTION**: Structs that contain only public member variables. In this case, no prefix and lower camelCase.

```
int thisIsAVariableInAStruct;
```

- C# Only: In the case where you have a public member variable in an internal class, then use lower camelCase without the `m_` prefix.

```
int thisIsAMemberVariable;
```

- Global variables are prefixed with `g_`, rest is upper CamelCase.
- Static variables are prefixed with `s_`, rest is upper CamelCase.

```
int g_ThisIsAGlobalVariable;
int SomeClass::s_ThisIsAClassVariable = 0;
```

- Constant variables are prefixed with `k`, the rest is upper CamelCase.

```
const int kFixedNumber;
```

Property names (C#)

- Property names use lower camelCase.
  Rationale: Unity used lower camelCase for properties since the beginning for whatever reason and now we have to stick to that. A lot of properties in our API that are very commonly used by our users are identical to class names, except for the lowercase first letter. Examples: gameObject vs. GameObject, transform vs. Transform, and there are lots more like that. Thus we can't simply rename the properties to upper CamelCase; we'd have to come up with a whole new naming scene that users would have to learn. Combine it with the fact that no users have complained about the properties being lower camelCase, and nooone have requested it to be changed, and it's easy to conclude that changing it would inflict more pain to our users than keeping it how it is. We use the same convention (lower camelCase) for properties that are not in our public API, because using a different convention for public and internal properties would be more confusing than just sticking to one convention consistently.

```
public float time { get { return m_Time; } set { m_Time = value; }  }
```

Bool property name considerations

- We try not to use "`is`" or "`use`" where possible:
    - kinematic instead of isKinematic
    - readyToPlay instead of isReadyToPlay
- We use "`use`" only where it is necessary and makes sense
    - useGravity
- We really try to avoid "`is`" but use it if it *really really*makes sense:
    - Application.isPlayer

Enumeration names (C++, C#)

- In C++ prefix enum names with `k`, the rest is upper CamelCase.

```
enum
{
     kDefaultWrapMode = 0,
     kClamp = 1 << 0,
     kRepeat = 1 << 1,
     kPingPong = 1 << 2,
     kClampForever = 1 << 3
};
```

- In C# both enum type names and enum value names are upper CamelCase.

```
public enum WrapMode
{
     Default = 0,
     Once = 1,
     Loop = 2,
     PingPong = 4,
     ClampForever = 8
}
```

Namespace names (C++, C#)

- C++ namespaces use all lowercase names.
- C# namespaces use upper CamelCase.
- Namespace names must be short.

Type alias names (C++)

- Typedefs always use upper CamelCase.

Template parameter names (C++)

- Names use upper CamelCase.
- `T` may also be used if it is the only template parameter.

Macro names (C++)

- `#define` names are in `ALL_CAPS`. Multiple words separated by underscores.

Code Formatting

- Basics
- Curly braces (C++, C#, Perl)
- Spaces (C++, C#, Perl)
- Vertical spaces (C++, C#, Perl)
- Line wrapping (C++, C#, Perl)
- Indentation (C++, C#, Perl)
    - C++ specific indentation
- Type formatting (C++)
- Class layout (C++)
- Class layout (C#)
- Capitalization (Perl)
- Parentheses (Perl)
- Inlining (Perl)
- Various

Basics

- Use tabs. One tab is four spaces.
    - Visual Studio users: for C# files it defaults to spaces, *so change it*!
      Tools->Options, Text Editor->C#->Tabs->Keep Tabs.
- Use Unix (LF) line endings in source files. (Visual Studio users can use Strip'em add-in.)
- Use one declaration per line (unless none of the variables are pointer or reference type).

```
int* a;
int& b;
int c, d;   // None of the variables are pointers or references
```

Curly braces (C++, C#, Perl)

- Braces must be aligned so the braces are at the same scope as the statement that proceeds them and the code within the braces is indented one tab level.
- Braces are always on a line by themselves.

```
void DoSomething ()
{
    if (x != y)
    {
        y = x;
    }
    else
    {
        x = y;
    }
}
```

- Braces may be omitted where the syntax allows it.
  The remaining content must still be indented one tab level and placed on a new line.

  However, braces should only ever be omitted for a single level of indentation. To improve readability, multiple levels of nested code should always use curly braces, except for the deepest level, where they can be omitted:

```
void DoSomething ()
{
    if (x != y)
        y = x;
    else
        x = y;

    if (x != y)
    {
        if (y != z)
            x = z;
    }
}
```

Spaces (C++, C#, Perl)

- Add spaces before and after binary and tertiary operators, including assignment operators:

```
a = b * c;
a *= b;
a = b ? c : d;
```

- Do not add a space after a unary operator:

```
a = !b;
a = -b;
```

- Add a space before an opening parenthesis and after a closing parenthesis, including after the flow control keywords `for`, `while`, `if` and `switch`:

```
void MyFunction ()
{
    if (a > b)
        a = b * (c + d);
}
```

- However do not add space between parentheses:

```
if (EditorGUI.EndChangedCheck ())
```

- Add space after `,` but not before in `for` sequences and arguments.
- Add space after `;` but not before in for expressions.
- Do not add spaces around `;` when used to end statements.

```
int j = 0;
for (int i = 0; i < 10; ++i, j += 3)
    MyFunction (i, j);
```

- Don't add spaces around `.`, `->`, `->*` and `.*` operators.

Vertical spaces (C++, C#, Perl)

- Multi-line type declarations must be separated by two blank lines:

```
struct MyStruct
{
};


class MyClass
{
};
```

- Add line break after `template<...>` in C++:

```
template<class T>
void TemplateFunction ()
{
}
```

- Add line break after attributes in C#:

```
[System.Serializable]
public class SampleClass
{
}
```

- When appropriate, separate code blocks by one empty line:

```
if (x == 0)
    return '?';

int functionResult = f (x);
char outputCharacter = functionResult + '0';

return outputCharacter;
```

- Group similar declarations and separate the groups by one empty line:

```
typedef signed char int8;
typedef signed int int32;

int32 FunctionOne (int8 x, int8 y);
int32 FunctionTwo (int8 x);
```

- Group related member functions and separate the groups by one empty line:

```
class SomeClass
{
public:
    SomeClass ();
    ~SomeClass ();

    int GetX ();
    int GetY ();

    void SetX (int x);
    void SetY (int y);

private:
    int m_X;
    int m_Y;
};
```

Line wrapping (C++, C#, Perl)

- Avoid breaking lines whenever possible.
    - When necessary, indent the remaining part one tab level.
- When necessary, break lines after boolean operators in conditional expressions:

```
if (time > startTime &&
    time < endTime)
{
    MyFunction (time);
}
```

- When necessary, break lines after `;` in `for` statements.

```
for (int i = 0;
    i < 10;
    ++i)
{
    MyFunction (i);
}
```

- When necessary, break lines after `,` in function calls and put each argument on its own line.

```
MyFunction (
    SomebodyElsesFunctionWithAReallyLongName (),
    AnotherReallyVerboseGuyWhoJustDoesntKnowWhenToStopTyping ());
```

- When necessary, break function declarations over several lines with each argument on its own line.

```
void MyFunction (
    int x,
    int y)
{
}
```

- Break initializer lists over several lines with each initializer on its own line.
- Keep the separating `,` at the beginning of the line.
  Rationale: Eases `#ifdef` in initializer lists.

```
Vector::Vector ()
:   m_X (0)
,   m_Y (0)
,   m_Z (0)
{
}
```

Indentation (C++, C#, Perl)

- Indentation always starts at column 0.
- Namespaces don't add indentation levels.

```
namespace
{

class MyClass
{
};

}
```

- The content of a scope must be indented one tab level relative to the parent scope.

```
{
    x = y * z;
    // ...
}
```

- Both case labels and code blocks are indented for switch statements.
  Rationale: The content of both cases with and without scope braces still line up at the same indentation level, like any other flow control statements.

```
switch (x)
{
    case 0:
        x = 1;
        break;
    case 1:
    {
        x = 2;
        break;
    }
}
```

- The else in a conditional statement is placed on the same indentation level as the if.

```
if (x == 0)
    y = 0;
else
    y = 1 / x;
```

- If an else condition in a conditional statement is followed solely by one new conditional statement, the new conditional can be placed on the same line as the else and the extra indentation level can be omitted to make the series of conditional statements read more similar to a switch statement.

```
if (x == 0 && y == 0)
    z = 0;
else if (x > 0)
    z = 1;
else if (y > 0)
    z = 2;
else
    z = 3;
```

C++ specific indentation

- In C++, the access modifiers in class declarations are not indented.
  Rationale: Keeping the access modifiers at a different indentation level to the actual declarations makes it faster to scan a class declaration to find the section you are interested in.

```
class MyClass
{
public:
    MyClass ();
    ~MyClass ();

private:
    int m_X;
};
```

- In C++, the # in preprocessor directives is never indented and is always placed at column 0.
  Rationale: Some compilers may accept the # placed at an arbitrary column, however this is not standard behaviour.
- Place whitespace between # and the keyword if necessary.
  Rationale: Allowed by standard.

```
#if UNITY
#   define FANTASTIC
#endif
```

Type formatting (C++)

- In C++, place the pointer and reference type modifiers with the type name.
  Rationale: Only one variable per line is preferred, so we can keep these things together without causing confusion.

```
int* pointerToInt;
```

- In C++, place const modifiers before type name (except when the pointer variable must be const)

```
const int* pointerToConstInt;
int* const constPointerToInt;
const float kGravity = 9.81f;
```

Class layout (C++)

- Class content must be in the following order: Public functions, protected functions, private functions, protected data members, private data members.
  Rationale: Header files serve as API documentation. This order ensures the things that are most important to other programmers come first. The least important implementation specific details come last. Ideally these should be put in the `.cpp` file, however this is not always possible in an easy manner.
  It would make sense to have all protected functions and variables before all private functions and variables. However, when trying to understand a new class it is often more helpful to have the state (the variables) all in one place, despite their access level.
- Unity specific macros must be placed at the top of the class declaration.

```
class WindZone : public Behaviour
{
public:
    REGISTER_DERIVED_CLASS (WindZone, Behaviour)
    DECLARE_OBJECT_SERIALIZE (WindZone)

    // ...
};
```

- Group the declarations in the following order within the different access level sections of the class.

```
class MyClass
{
public:
    typedef int MyType;

public:
    MyClass ();
    // ...
};
```

- Type declarations (only if part of the class interface)
- Constructors and destructor
- Operators
- Functions grouped by topic (for instance functions handling rotation in various ways, functions handling translation)
    - Accessors
    - Mutators
- To help readability you may repeat the access modifier to separate these groups, even if it is superfluous.

- Move inline functions away from the class interface declaration, except for empty implementations.
  Rationale: Don't clutter the class interface declaration with implementation details.

```
class MyClass
{
public:
    MyClass (int x);
    MyClass (const MyClass& that);
    ~MyClass () {}

    MyClass& operator= (const MyClass& that);

    inline int GetX () const;

protected:
    void SetX (int newX);

private:
    // Any private functions

protected:
    // Any protected variables

private:
    int m_X;
};


inline int MyClass::GetX () const
{
    return m_X;
}
```

- Implementations that are very short, typically one statement for variable getters and setters may be kept in the class declaration.

```
class MyClass
{
public:
    // ...
    inline int GetX () const        { return m_X; }
    inline float GetForce() const   { return m_Force; }

    // ...
};
```

- Consider aligning the setter and getter bodies at the same tab level to improve readability. The function signature is the most important part.

Class layout (C#)

- Class content must be in the following order: Public data members*, protected data members, private data members, public properties, protected properties, private properties, functions.
  Rationale: It should be easy to quickly get an overview over the state that a class maintains, so all data members should be together at the top before all functions.
  * Note that there should never be any public exposed data members in classes that are in our public API.
- Properties that are directly linked to a member variable may be listed above that variable.

```
public float time { get { return m_Time; } set { m_Time = value; } }
internal float m_Time;
```

- We don't seem to have any convention for the order of public, protected, and private functions.

Capitalization (Perl)

- Use Camel Case for everything (including subroutine names, variable names, module names, and filenames).
- Capitalize the first letter of subroutine, module, and file names (`build.pl` being the one exception to this rule). This especially helps make it easy to recognize subroutines defined by us, as opposed to subroutines provided by standard Perl modules.

```
sub MySubroutine1
{
    my $fullName = "my full name";
    my $shortName = "my short name";
    MySubroutine2 ($fullName, $shortName);
}
```

Parentheses (Perl)

- Always use parentheses when calling subroutines

```
MySub ($fullName, $shortName) or croak ("Subroutine MySub failed");
```

- Do not use parenthesis in subroutine declarations regardless if arguments are required

```
sub MySubWithNoArgs
{
}

sub MySubWithArgs
{
    my ($arg1, $arg2) = @_;
}
```

Inlining (Perl)

- Do not put if statements at the end of a line; use C-style if statements
- If the body of the statement is sufficiently short, it is allowed to put the parenthesis and body on the same line as the if statement

```
# Incorrect
$foo =~ s/$old/$new/ if $myBool;

# Correct
if ($myBool)
{
    $foo =~ s/$old/$new/;
}

# Also Correct
if ($myBool) { $foo =~ s/$old/$new/; }
```

Various

- Do not use "this" (`this->m_Member` or `this->SomeMethod ()`), except where it is really needed. Member variables already have `m_` prefix, so adding "this" just increases noise.
- Do not add comments to files like "created by Foo Bar on 2009/01/01, copyright Bar Foo". Some IDEs (XCode) add that automatically - please remove them.
- Always `use strict` and `use warnings` in Perl files.

Design Considerations

Functions (C++, C#)

- Prefer early-out return statements to deeply nested if/else constructions.
- Keep functions short and to the point.
    - Prefer functions that do one definable thing.
    - Prefer short functions (20-40 lines) to longer ones.
    - Refactor large functions that perform several distinct tasks into several smaller ones.
      Rationale: Large functions that perform several distinct tasks make it harder to understand and confidently modify new code.

C++

- Prefer const reference input arguments to const pointer input arguments.
    - Use const pointer input arguments when a NULL argument makes sense.
- Use pointers instead of references for output arguments.
- Mark all functions that do not modify state (including accessors) `const`.
- Input arguments come before output arguments.

```
void CrossProduct (const Vector3f& v1, const Vector3f& v2, Vector3f* result);
```

Constants and enums (C++, C#)

- Avoid hard-coded constants.
- Prefer enums declared in a class or otherwise scoped.
- Prefer enums for constants.
- Put each enumeration value on its own line.
    - If only defining a single enum, such as a constant, putting the entire declaration on a single line is preferred.

```
enum { kMyConstant = 123 };
```

- Use bit shift expressions when declaring flags (only in C++?):

```
enum
{
     kDefaultWrapMode = 0,
     kClamp = 1 << 0,
     kRepeat = 1 << 1,
     kPingPong = 1 << 2,
     kClampForever = 1 << 3
};
```

Classes (C++)

- If a class needs a destructor, copy constructor or assignment operator, it must provide all of them. If any of these are not desirable,

make them private and do not supply an implementation.
Rationale: If these are not supplied, the compiler will happily provide a crappy one for you.
    - In Unity code, making the copy constructor and assignment operator private is accomplished by deriving from `Object` (or from something that derives from `Object`). It may also be accomplished explicitly by deriving from `NonCopyable`.
- When overriding a virtual function, always specify the `virtual` keyword.
- Prefer that subclasses keep a "is-a" relationship with the super class.
    - Consider using composition when the relationship is better modeled as "has-a".
- Group related classes as a unit in a directory and prefer tight encapsulation.
    - Consider using friends to enhance encapsulation.
      View friends as the C++ equivalent of package/assembly access level.
    - Avoid cyclic dependencies between these units.

Namespaces (C++)

- Do not, under any circumstances, use `using namespace` at a global level. Not in header files, not in implementation files.
- Restrict use of `using namespace` to well defined scopes, like functions.

```
void MyFunction ()
{
    using namespace std;

    // Code using the std namespace ...
}
```

Tutorials

Create a new Component to Unity in C++

Part 1: Create a New Behaviour

First of all we need to make the basic C++ class. As an example we will make a new class called *MyBehaviour*

The Header File

- Start by creating a new folder as followed: *../Runtime/MyCode*
- Create a new header file to that folder named as: *MyBehaviour.h*

First we add pragma guard and basic include to the header file:

```
#pragma once
#include "Runtime/GameCode/Behaviour.h"
```

RTTI (run-time type information) setup

`REGISTER_DERIVED_CLASS (MyBehaviour, Behaviour)` is macro that sets up Unity's RTTI system. You need to specify the class itself and its parent class.

`DECLARE_OBJECT_SERIALIZE (MyBehaviour)` declares the necessary functions for the serialization system.

```
class MyBehaviour : public Behaviour
{
 public:
 REGISTER_DERIVED_CLASS(MyBehaviour, Behaviour)
 DECLARE_OBJECT_SERIALIZE(MyBehaviour)
```

The remaining code should be fairly self explanatory. AddToManager is a useful callback provided if you inherit from Behaviour. An Object is added to a Manager when it is enabled and its GameObject is active. Otherwise it will be removed from the Manager.

```
 MyBehaviour (MemLabelId label, ObjectCreationMode mode);

 virtual void Reset ();
 virtual void Update ();
 virtual void AddToManager ();
 virtual void RemoveFromManager ();

    void SetSpeed (float speed) { m_Speed = speed; }
    float GetSpeed () const { return m_Speed; }
private:
 float m_Speed;
 BehaviourListNode m_UpdateNode;
};
```

Implementation (.cpp) File

Create a new source file next to the header named as: *MyBehaviour.cpp*

First we add some basic includes to the file:

```
#include "UnityPrefix.h"
#include "Runtime/Serialize/TransferFunctions/SerializeTransfer.h"
#include "Runtime/BaseClasses/IsPlaying.h"
#include "Runtime/Graphics/Transform.h"
#include "MyBehaviour.h"

using namespace Unity;
```

RTTI setup

After having declared serialization and RTTI in the header we have to put in the implementation of those subsystems in the C++ file.

```
IMPLEMENT_OBJECT_SERIALIZE (MyBehaviour)
IMPLEMENT_CLASS (MyBehaviour)
```

Serialization

Serialization in Unity is extremely powerful, once you understand how to use it.

Unity's serialization system handles backwards compatibility completely transparently. You can reorder properties at your hearts content without losing data. You can add new properties which will simply be ignored if an older file is being loaded. New values added, will be left untouched and can thus be initialized in the construct or the Reset function.

You can register converters and version the serialized data arbitrarily. On top of that data is stored in optimized binary formats which also store the necessary type information in order to restore backwards compatibility. There are multiple serialization template classes. Optimized stream readers, which are as fast as just memcpy'ing data and there are safe reading functions which can load all data by name and perform conversions on the serialized data.

Best of all, you have to think about nothing of this, you just have to implement the Transfer function by transferring all properties you want to serialize. This gives a perfect combination between speed, ease of use and backwards compatibility.

Unity can serialize Pointers to objects which can live anywhere (e.g. in the same scene or in different files) without much fuzz.

Serialization is generally handled in 2 functions: Transfer and AwakeFromLoad.

```
template<class TransferFunction>
void MyBehaviour::Transfer (TransferFunction& transfer)
{
 Super::Transfer (transfer);
 TRANSFER (m_Speed);
}
```

Add something about UpdateNode

```
MyBehaviour::MyBehaviour (MemLabelId label, ObjectCreationMode mode)
: Super(label, mode)
, m_UpdateNode (this)
{}

MyBehaviour::~MyBehaviour ()
{}

void MyBehaviour::Reset ()
{
 Super::Reset ();
 m_Speed = 0.1F;
}

void MyBehaviour::Update ()
{
    // C++ behaviour get updated in playmode and edit mode.
    if (!IsWorldPlaying())
  return;
 Transform& transform = GetComponent(Transform);
 transform.SetPosition(transform.GetPosition() + Vector3f(0, m_Speed, 0));
}

void MyBehaviour::AddToManager ()
{
 GetBehaviourManager ().AddBehaviour (m_UpdateNode, -1);
}

void MyBehaviour::RemoveFromManager ()
{
 GetBehaviourManager ().RemoveBehaviour (m_UpdateNode);
}
```

Part 2: RTTI (run-time type information)

When declaring a new class there is a process that must be followed.

To use a class we first need to assign it an ID number. Every class is registered by an integer value. This is used by the serialization system to find class prototypes, and allows you to rename a C++ class while still making it load older versions. It is also used for our very fast RTTI system.

We recommend that source code customers use IDs in 500-700 range in order to avoid conflicts. In the case where a conflict does exist, the editor will inform you of the conflict.

When creating a new class ID, open *ClassID.h* which has all Class IDs in one place.

ClassID.h

```
DefineClassID (MyBehaviour, 222)

kLargestRuntimeClassID,
```

Second, the class must be registered in the *ClassRegistration.cpp*

ClassRegistration.cpp

```
void RegisterAllClasses()
{
...
 REGISTER_CLASS (MyBehaviour)
...
}
```

Part 3: Script Binding and JAM

Script Binding

The basic idea behind the .txt files is to make it easy to mix C++ and C# code in the same file.

It isn't necessary to understand the whole binding process at this point, but you should take a quick look at the documentation: Adding Script bindings to a C++ class

Create a new file to the *Runtime/Export* folder called *MyBehaviourBindings.txt*

Lets start by defining C++ and C# includes.

```
C++RAW
#include "UnityPrefix.h"
#include "Runtime/Mono/MonoExportUtility.h"
#include "Runtime/MyCode/MyBehaviour.h"

CSRAW
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Collections;
```

Expose **MyBehaviour** class and a custom property **message** to the C# side. **message** is defined here as a property with get and set functionality.

```
namespace UnityEngine
{
 CLASS MyBehaviour : Behaviour
  CUSTOM_PROP float speed { return self->GetSpeed(); } { self->SetSpeed(value); }
 END
CSRAW
}
```

This generates C++ code from the .txt file. The generated function must be registered with mono, so that when the speed property is accessed it will call the following auto-generated C++ code.

```
// Example of auto-generated C++ code from the .txt file
void CUSTOM_get_speed (ReadOnlyScriptingObjectOfType<Animator> self, float speed) {
return self->GetSpeed(); }
```

In order to register the generated code, you need add one registration function per .txt file. The function is found in the auto-generated cpp file.

You need to add the registration function to MonoICallRegistration.cpp, otherwise you will get a Missing Method exception when calling the function from C#.

```
void ExportMyBehaviourBindings (); <<<

 ....

static InternallCallMethod* sMonoBindingsRegistration[] =

    &ExportCursorBindings,

 &ExportMyBehaviourBindings, <<<

 #if UNITY_XENON_API
    &ExportXboxServices,
    &ExportXboxKinect,
    &ExportXboxAvatar,
```

JAM

To add MyBehaviour source and binding files to the build, they have to been added to the jam files.

Open */Projects/Jam/RuntimeFiles.jam* for editing

Add following line to the `RUNTIME_EXPORT_TXT_SRCS =`

```
Runtime/Export/MyBehaviourBindings.txt
```

Go down the file and add these lines somewhere after the runtimesources is defined and initialized.

```
runtimesources +=
  Runtime/MyCode/MyBehaviour.cpp
  Runtime/MyCode/MyBehaviour.h
 ;
```

Part 4: Expose to Editor

"Add new component"

By default components show up in the Component/Misc menu. To override and add them to a different menu, you have to add them to *ComponentRequirement.cpp*. You can define required components for them, and make them be a part of a component group.

Open *ComponentRequirement.cpp* and add following lines to InitComponentRequirements()

```
void InitComponentRequirements ()
{
...
 AddRequiredClass (MyBehaviour, Light);
...
 AddGroup ("MyCode");
 AddToComponentHierarchy (MyBehaviour);
...
}
```

Custom Inspector

Part 5: Custom Inspector

Inspector

- csharp
  - MyBehaviourEditor
- cpp
  - ResourceManager.cpp

Part 6: Bind to gameObject

gameObject member variable

- UnityEngineGameObject.txt - same
  - common_include
  - remap

Adding Script bindings to a C++ class

Unity uses a perl script to generate script bindings. We use definition files which have to have **.txt** extension. From the **.txt** file we generate a C++ binding, a C# interface, and Script Reference documentation.

If you create a new script binding **.txt** file, you have to place it in the Runtime->Export folder. To get started, we recommend you to add a class to an existing **.txt** file to get accustomed to the process.

Classes in script bindings usually inherit either from Component or Behaviour. In the case of assets you inherit directly from Object.

The basic idea behind the **.txt** files and cspreprocess is to make it easy to mix C++ and C# code in the same file. To run it use: cd into the unity directory then perl Tools/cspreprocess.pl

```
CLASS ConstantForce : Behaviour
    /// The force applied to the rigidbody every frame.
    CUSTOM_PROP Vector3 force { return self->m_Force; } { self->m_Force = value; }

    /// The force - relative to the rigid bodies coordinate system - applied every frame.
    CUSTOM_PROP Vector3 relativeForce { return self->m_RelativeForce; } { self->m_RelativeForce = value;
}

    /// The torque applied to the rigidbody every frame.
    CUSTOM_PROP Vector3 torque { return self->m_Torque; } { self->m_Torque = value; }

    /// The torque - relative to the rigid bodies coordinate system - applied every frame.
    CUSTOM_PROP Vector3 relativeTorque { return self->m_RelativeTorque; } { self->m_RelativeTorque =
value; }
END
```

This is actually C++ code that will be executed (two implementations, for the getter and the setter).

```
CUSTOM_PROP Vector3 force { return self->m_Force; } { self->m_Force = value; }
```

Self always refers to the object itself:

```
   { return self->m_Force; } { self->m_Force = value; }


CUSTOM string DoSomething (string inputValue)
{
    string returnValue = self->DoSomethingWithString(MonoStringToCpp(inputValue));
    return MonoStringNew (returnValue);
}
```

The most common binding keywords are CUSTOM, CUSTOM_PROP, AUTO and AUTO_PROP.

```
CUSTOM void DoSomething (float input) { self->DoSomething(input); }

CUSTOM static void DoSomething (float input) { DoSomethingGlobal(input); }
```

For simple functions which use the same parameters in C++ you can simply use AUTO

```
AUTO void DoSomething (float input);
```

- Passing objects by reference. Eg. anything inherited from Object is done automatically via the Reference template class.
- Built-in types, like float, int, double, Vector3, Vector4, Vector2 are also handled automatically.
- Strings however must be converted manually at the moment:
    - On the C++ side, MonoString* will be passed in, you can easily convert it to a C++ string with MonoStringToCpp. Return values on the C++ side are passed as MonoString* too, thus you have to convert them using MonoStringNew.

When writing a lot of bindings, sometimes you discover a need to remap a type from the C# name to another name in C++. This is done using the typemap which is defined in: Runtime/Export/common_include

The syntax is fairly straightforward. For example:

```
Vector3=>Vector3f
```

NOTE:

This is especially important for C++ binding code that wants to install a native pointer into a managed script object. By default, pointers within the binding layer are represented as ReadOnlyScriptingObjectOfType<T> that expose their GetScriptingObject() method only in the editor.

To switch that to ScriptingObjectOfType<T>, use "[Writable]" on your type like so:

```
CUSTOM private static void Internal_Create ([Writable]MyClass self)
 {
  MyClass* obj = NEW_OBJECT_MAIN_THREAD (MyClass);
  obj->Reset();
  ConnectScriptingWrapperToObject (self.GetScriptingObject(), obj);
  obj->AwakeFromLoad (kInstantiateOrCreateFromCodeAwakeFromLoad);
 }
```

and then go to Runtime/Mono/common_includes and add a mapping for this type:

```
[Writable]MyClass=>ScriptingObjectOfType<MyClass>
```

Creating a new script binding .txt file:

When creating a new .txt file, please use this as a template.

```
C++RAW
#include "UnityPrefix.h"
#include "MonoExportUtility.h"
#include "BaseScript.h"

CSRAW
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Collections;


namespace UnityEngine
{

CLASS MyNewClass : Component
// ...your class script interface here...
END


CSRAW
}
```

Using generated C++ files and updating UnityEngine.dll

From a script binding file `Something.txt` a C++ file `Generated/Something.cpp` is generated. Whenever you add new .txt files, you **mu st** add the generated C++ file to all the projects (XCode, CodeWarrior and both Visual Studio projects). Otherwise you'll get errors when starting a game (Mono won't be able to find functions).

Whenever you made significant changes to the script bindings, you need to transfer generated C++ files and compiled `UnityEngine.dll` t
o a Windows development machine. If you use a build machine, no extra steps are needed; but if you do building manually, here are the steps:

1. After making changes to .txt files, run Unity editor. It will regenerate C++ files and recompile `UnityEngine.dll`.
2. Transfer `Runtime/Export/Generated/*.cpp` and `build/UnityEngine.dll` to the windows development machine.
    a. For standalone Windows games, run `windowsPrepare.cmd` on Windows development machine and recompile.
    b. For windows web player, either regenerate the installer, or copy `build/UnityEngine.dll` to web player's installation folder.

A quick reference overview of features supported by cspreprocess:

```
CLASS [name], STRUCT [name], SEALED_CLASS [name], END

Example:
    CLASS Foo
        ... other tags ...
    END

 ENUM {public|private}? [name] ... END
  Behaves as CSRAW, replacing the ENUM [name] part with public/private enum .... following lines ...
  Can be nested within classes and structs. Defines a separate class in the documentation.
  Example:
    ENUM Bar
     /// Docstring...
     baz = 1,
     .....
    END

 DOCUMENT [title] {[filename]) ... END
  Generates a file document that does not relate to a class or a struct. Useful for adding extra
documentation to the script reference.

 AUTO [function signature;]
  Creates the CS and C++ wrapper for the member function with the signature.
  Example:
    AUTO void Create (Vector3 pos);

 AUTO_PTR_PROP [Type] [propertyName] [GetName in wrapped C++ class] [SetName in wrapped C++ class]
  Generates a property from a getter and setter for a pointer
   Example:
    AUTO_PTR_PROP Light castlight GetLight SetLight

 CSRAW [raw c# code, multiple lines ok]
  Pass through C# code, multiple lines if needed.
  An attempt is made to parse the code, so most things should get detected.
  Note this is exceptionally brittle, so take care to keep it simple and check that
documentationactually gets generated correctly.

 C++RAW
  Pass C++ through without further parsing.

 CSNONE [function signature;]
  Inserts this into documentation as Overridable Methods (in MonoBehaviour)
  or Messages Sent (elsewhere), without actually generating any code.

 CUSTOM [public|private|internal] [static] <c# method or constructor signature> <c++ function body>
  Generates a c# method or static method which has the c++ function body as its definition.
  public is optional and the default access level
  Example:
    CUSTOM void Foo (Bar bar) { self->Foo(bar); }

 CUSTOM_PROP [static] <c# property sig> <c++ getter body> [<c++ setter body>]
  Note CUSTOM_PROP is currently always public. Use methods for private or internal accessors.


 OBSOLETE <error|warning|planned> <text>
   Adds an [Obsolete ...] attribute to the following method or class and hides the class from the docs.
   If mode is error, the obsolete attribute will generate an error if the feature is used.
   If it is warning, the user will only get a compile time error.
   If it is planned, the feature is only removed from the docs, but no obsolete attribute is added
```

Allocating memory

Temporary Allocations

Memory that lives for less than one frame must use temp allocations. Temp allocations are very fast and reuse previous temp allocations effectively. Not doing this is a massive waste. It leads to fragmentation and bad performance. It is unacceptable to write new code without using temp allocations properly.

```
// use kMemTempAlloc allocator label to get the fast temporary allocator
dynamic_array<UInt8> myTempArray (kMemTempAlloc);

{
   // Allocate a floatArray of 12 floats automatically deallocates when leaving the
{} scope
   float* floatArray;
   ALLOC_TEMP (float, floatArray, 12);
}
```

Allocating using UNITY_NEW and UNITY_MALLOC macros : Labels

Allocations can now have a label attached, which serves 2 purposes:
- Registering the allocation in the MemoryProfiler.
- Routing that allocation to the allocator bound to that label.

Allocating blocks of memory

All allocations should be made with either
UNITY_MALLOC
UNITY_REALLOC
UNITY_MALLOC_ALIGNED or
UNITY_REALLOC_ALIGNED
and freed using
UNITY_FREE
UNITY_FREE_GENERIC (will look up the corosponding label)

Temporary memory should be allocated using the macros:
ALLOC_TEMP / ALLOC_TEMP_ALIGNED / MALLOC_TEMP
These do not require deallocation, since the memory is placed on the stack. If the allocation is larger that 4K the tempoarary allocator will be used, and the memory is auto freed when leaving scope:

Newing and deleting objects

There is also a macro for newing objects:
UNITY_NEW
UNITY_DELETE
All new and delete are overwritten and assigned the label kMemNewDelete.
This is where 90% of our news go at the moment. This is not very informative in the MemoryProfiler, but slowly we'll move news to using the macro.

Where do the allocations end up

The map between labels and allocators can be different on the platforms, but all are set up in MemoryManager.cpp
Right now, there are not that many different kinds of allocators in play, but this will change in time.
The 2 main allocators that are used are DynamicHeapAllocator and UnityDefaultAllocator(unsing normal malloc as underlying allocator)
For the temporary allocations the StackAllocator is used. The main advantage of this allocator is that it is fast and lockless.

One way to make objects or std containers use the temporary allocator is by using the kMemTempAlloc label

```
UNITY_VECTOR(kMemTempAlloc,CharacterInfo) tempCharacterInfo;
```

If the stack allocator is not empty at frame end, it will trigger an assert.

Lockless allocations

The main allocator is also lock free when being called from the main thread (as are the temp allocators). This is done by having a TLSValue containing a non locking allocator for the main thread, and all threads pointing to the same locking allocator. if allocations made on the main thread are deleted from a thread, the pointer is simply put on a queue, and deleted later from the main thread.

One thing that should be considered is that Jobs running on threads should not have to allocate any memory. Memory blocks for input/output and workbuffer, can easily be supplied by the main thread.

Longer running threads might have memory passed from the main thread, that is then deleted by the thread. Either the normal approach of having the memory added to a main thread queue can be used, or assigning a different - locking allocator, and the memory can be safely allocated on one thread and deleted on another.

Allocation tracking and memory hierarchy

With memory tracking, we now have the ability to nest allocations under root allocations. This is done on a per thread stack basis, where allocation owners can be pushed and popped.

Objects can now be created with the macro

```
object = UNITY_NEW_AS_ROOT(GameObject, kMemBaseObject, NULL, NULL);
```

which will make the object returned by new, own all allocations made inside the constructor of the object.

Root allocations that are not gameobjects should be registered with a 3rd argument and is then put in a list of other roots. These are collected and displayed by the memory profiler.

```
manager = UNITY_NEW_AS_ROOT(MyManager, kMemManager "MyManager.Displayname", "");
```

Furthermore, all containers declared with the UNITY_VECTOR/MAP/STRING etc will know that the returned object is the owner of its data, and allocations or reallocations caused by f.ex. push_back will count towards the objects memory usage. also dynamic_array knows of its original owner.
This has the advantage, that we can now ask an object how much memory it uses - including vectors, arrays and strings. The memory usage of the object can be queried by:

```
GetMemoryProfiler()->GetRelatedMemorySize(object);
```

This is used for BaseObject::GetRuntimeMemorySize, which is now generic (and precise) for all (most) objects. The place where this deviates is when driver memory is in play. There are functions on the memoryprofiler to manually register memory to an ID. This results in a few classes that needs a custom GetRuntimeMemorySize like f.ex Texture:

```
int Texture2D::GetRuntimeMemorySize() const
{
#if ENABLE_MEM_PROFILER
 return Super::GetRuntimeMemorySize() +
  GetMemoryProfiler()->GetRelatedIDMemorySize(m_TexID.m_ID) +
  (m_UnscaledTextureUploaded?
  GetMemoryProfiler()->GetRelatedIDMemorySize(GetUnscaledTextureID().m_ID):0);
#endif
 return sizeof(Texture2D);
}
```

This is not something you need to use in normal workflow, since it is build into baseobject produce. If it is necessary to set the owner manually, there is a macro SET_ALLOC_OWNER(ptr) to set allocation ownership for the current scope. This is useful if for example a GameObject constructor calls outside its own scope to a singleton that allocates memory for itself. Then the Singleton function could start out by declaring itself owner of the allocations to follow.

The MemLabelId can now also contain an owner. The Gameobjects have the MemLabel on the class, and this label can be passed when making allocations. This removes the overhead of asking the TLS stack for the current owner.

Sometimes it can be useful to be able to transfer a piece of memory to another owner, and that can be done with UNITY_TRANSFER_OWNERSHIP.

There are also getters, that can get the current set owner, or the owner of a specific ptr.

```
GET_CURRENT_ALLOC_OWNER()
GET_ALLOC_OWNER(ptr, label)
SET_ALLOC_OWNER(owner)
UNITY_TRANSFER_OWNERSHIP(source, label, related)
```

Whenever an object is deallocated, the memoryprofiler will check that all memory related to this pointer has also been deleted. This is useful for detecting leaked memory.

If not set up correctly, memory that is not related to an object might get related to it anyway (as in the Singleton example above).

Initializing and cleaning up singleton managers

There are 2 methods in player.cpp: RuntimeInitialize and RuntimeCleanup
This should be the very first thing - and last thing that runs.
In RuntimeInitialize simple singleton managers can be allocated, and then destroyed in RuntimeCleanup.
At the moment almost no allocations are made before RuntimeInitialize, and in players, only about 30-50 allocations are left at cleanup. It is ofcourse the goal to get down to 0. In this way, leaks will be easy to detect.
If leaks should occur, there is the possibility to turn on stacktracing of all allocations, and a list of the offending allocations can be output at program exit.

One thing that should be considered is to not have static containers in functions or static manager objects that get initialized on app startup. So instead of using:

```
// Do not use managers like this
MyManagerWithContainers gManager;
MyManagerWithContainers& GetManager(){return gManager;}
```

Use a pointer to the manager and use RegisterRuntimeInitializeAndCleanup to call StaticInitialize/Cleanup. The order of Initialization methods registered this way is not fixed, but the cleanup calls are called in reverse order of the initialization calls.

```
MyManagerWithContainers* gManager;
StaticInitialize(){gManager = UNITY_NEW(MyManagerWithContainers, kMemManagers);}
StaticCleanup(){UNITY_DELETE(gManager, kMemManagers);}
MyManagerWithContainers& GetManager(){return *gManager;}

static RegisterRuntimeInitializeAndCleanup
s_MyManagerWithContainersCallbacks(StaticInitialize, StaticDestroy);
```

Or perhaps even use UNITY_NEW_AS_ROOT, to gather all memory of the object to be queried later.

Managing and Building Resources

Built-in Resources

How and Where

Unity has some "default resources": assets that are available even if not explicitly in the project (built-in shaders, box/sphere/cylinder/... meshes, default font, default particle texture etc.). Some default resources are part of Unity itself (not the game's data), so **back wards compatibility is very important**.

Pre-built resource files are in Mercurial, `External/Resources/builtin_resources/builds.zip`
Resources project is in the codebase: `External/Resources/builtin_resources`.

Workflow to modify them

1. Modify a file in the project, commit & push
2. Rebuild on TeamCity by running Build Builtin Resources configuration
3. **Inspect full Build Log** for diffs and check if any suspicious diffs are there

4. **If and only if no suspicious diffs** are there: take builds.zip from build artifacts and put it into `External/Resources/builtin_resources/builds.zip`
5. Commit builds.zip and push

If you want to add new resource (new builtin shader etc.)

1. Pick a place to add it under `External/Resources/builtin_resources`
    - `Assets/DefaultResourcesExtra` is preferred - this will make resource appear to be built-in, but when building a game, resource data will be included into game file.
    - Use `Assets/DefaultResources` otherwise, but **only if absolutely necessary**. Things that need to be available *no matter if anything in the game references them* or not go there (e.g. some internal shader that does deferred lighting calculations).
2. Modify `Runtime/Misc/ResourceManager.cpp` to include the new resource.
    - DefaultResourcesExtra go into InitializeExtraResources; DefaultResources into InitializeResources.
    - You'll need to come up with an integer ID for the new resource; which has to be unused by others and preferably similar to other similar resources.
3. Then do all steps as with regular resource modification workflow.

Locally test building the resources (while developing):

- As above, just replace anything with TeamCity with `perl Tools/build_resources.pl -builtin`
- This will print all diffs to stdout (redirect to file if needed)
- You'll need to have Editor build ready.
- Debug Editor build will fail because of breaking hard on asserts.
- On non-Windows this *will not* produce Xbox 360 & PS3 shaders!
- Use workflow above (with TeamCity) to actually commit resources!

See Also

Editor Resources

Standard Assets

Editor Resources

How and Where

- Editor resources project lives in the codebase: `External/Resources/editor_resources`.

Workflow to modify them

1. Build&Run a Debug build of Unity
2. Open and make changes to the editor resources project
3. Close Unity
4. run this to build resources:
   `perl Tools/build_resources.pl -editor`
   Or use the fancy new shortcut for this:
   `run b r e`
   (which is the shortcut for the long version: "run build resources editor")
5. Build editor since jam copies the resource files
6. Run Unity with a project other than editor resources to test the changes were built properly
7. Commit/Push (run via command line to get .hglf resource file committed) 'hg commit'

Editing Default Editor Resources

Editing Images

- Edit in Photoshop
- Use the Slice / Slice Select tool to create new slices for new icons
- Save by
    - using `File > Save for Web & Devices...`
    - Click `Save`
    - Make sure the images are saved in the correct folder
    - Choose Slices: `All Slices`
    - Click `Save`
- You should now get a `Replace Files` dialog - if not, you are probably saving in the wrong folder
    - Leave the checkmarks for all files and click `Replace`
- In the import settings for all new exported images, make sure that `Generate Mip Maps` is disabled and that the proper Texture Format is set (usually `ARGB 32 bit`).
- Once editing is done, remember to build the editor resources using the steps above in "Workflow to modify them"

Editing Styles

See the page Working with Editor Styles.

See Also

Built-in Resources

Standard Assets

Multithreading in Unity

Unity comes with a job system. It works, it's cross platform, it just needs more people using it!

This article is about showing some practical examples of how it can be used in Unity.

Here is an example of how it is used in Mecanim.

```
void Animator::UpdateAnimators ()
{
    JobScheduler& scheduler = GetJobScheduler();
    JobScheduler::JobGroupID jobGroup;

    size_t avatarJobCount = activeAvatars.size();
    jobGroup = scheduler.BeginGroup(avatarJobCount);
    for (size_t i = 0; i < avatarJobCount; ++i)
    {
        Animator& avatar = *activeAvatars[i];
        scheduler.SubmitJob (jobGroup, RetargetStepStatic, &avatar, NULL);
    }

    ….

    // Wait for all simulation to be completed
    scheduler.WaitForGroup (jobGroup);
}

void* Animator::RetargetStepStatic (void* userData)
{
    static_cast<Animator*> (userData)->RetargetStep();
    return NULL;
}
```

Here is some rules you should follow when multithreading code.

#1 Ensure that you have very clearly separated inputs and outputs. Never ever use global variables from the work threads.

Don't access other components from multithreaded code.

For example in Mecanim data is separated by Constant data (AnimationClips, StateMachine and BlendTree), Workspace memory (ControllerMemory) and Outputs.

Shared data is completely constant. const is used aggressively to ensure code does not violate this rule. Workspace memory is created once for every Animator (Thus we can safely multithread multiple Animators at the same time). Outputs are also created on for each instance. Basically Data Oriented Design (Links) greatly helps with writing safe multithreaded code.

#2 Enabling / Disabling multithreading should not affect any outputs. (Script callback order, order in which objects are rendered)

For example, performing frustum culling multithreaded by splitting the array of bounding volumes into multiple pieces should not change the order of visible renderers thus changing rendering order and the order of OnBecameVisible callbacks.

#3 Clear inputs & outputs

Don't assume, keep data structures straightforward and clearly isolated between objects. Never ever access global variables. Keep clear inputs and outputs for functions. If you don't know what state is being modified and if it might be shared globally or with other things executing on other threads, then you are doing it wrong. First and foremost get 100% clarity on what is being modified by who and where.

Example:

You might assume that calling Transform::CalculateTransformMatrix on transform components from multiple threads is safe. *It is not!*

m_CachedTransformMatrix is being written from Transform::CalculateTransformMatrix. You can easily end up in situations where multiple threads write to it at the same time in inopportune ways.

A common approach to multithreading is to extract scattered data like transform positions, game object layers etc before multithreading starts in a simple struct that is then passed to the multithreaded functions to consume. The multithreaded functions then operate on the provided data.

#4 Mutable shared state will always be slow

No matter how smart you are about not using mutex locks for reading & writing to shared state. It will never be fast. Reading and writing to shared state willl always lead to cache lines having to be synchronized between processors, which will always be slow. First and foremost always eliminate shared mutable state. Clearly seperated inputs & outputs help a lot. Ensure that each thread has one input & output. And make sure the data consumed by one thread is packed tightly together.

if code is thread safe without mutex locks that we are trying to avoid

#5 If you are using mutex locks you are doing it wrong!

We use mutex locks in unity to protect long running operations from race conditions. However when using the job system, if you find yourself having to use a mutex lock. Then you very likely to be doing something wrong. The example above using a isSimulating bool and WaitForGroup preferrable to mutex locks. Always aim to have your data so clearly separated that mutex locks are not required.

#6 Avoid global variables in multithreaded code

Global variables make it very difficult to understand

We have two common approaches to job based multithreading

#1 Run multithreaded wide and wait immediately (Simple approach) [Mecanim, Renderer culling in 4.1]

A simple first step to multithreading some code is to run it in parallel and wait for the computations to complete.

For example in Mecanim, we have to call script code at various stages during the animation update.

1. Evaluate RootMotion (Multithreaded)
2. Call OnAnimatorMove (Single threaded)
3. Perform FK & Retargeting (Multithreaded)
4. Call OnAnimatorIK (Single threaded)
5. Write outputs to Transform (Multithreaded)
6. Send messages to dirty affected components (Single threaded)

This is not optimal because none of the animation code is running while we invoke OnAvatarMove. Unfortunately script code consumes the evaluation generated by the multithreaded code. Thus other cores will be idle while calling script functions. Fortunately all the heavy code is fully multithreaded, while the script code is hopefully very fast.

Run multithreaded in paralell to scripting code. (Shuriken, Skinning, Multithreaded renderer)

For performance it is preferrable to run multithreaded code completely in parallel to scripting code, but naturally it is a lot more work to ensure that everything works consistently. You must handle all corner case situations when you take this approach, you have to write tests that stress each situation, because it lead to very hard to debug situations if we don't.

A couple of common things that a script can do while your multithreaded code is running in parallel:

- The object you are simulating is destroyed with DestroyImmediate
- The object you are simulating is deactivated while multithreaded code is running
- Properties on the object you are simulating are being modified.
- Results of the simulation are being queried while running

When taking this approach, extra care must be taken to protected the user modifying properties while multithreaded code is running. Engine code being multithreaded should not be visible to the user.

In Shuriken we run the particle simulation until it is consumed by either ParticeSystem.GetParticles or the renderer starts using it. Calling ParticleSystem.gravityModifier = 10; will automatically wait for all particle system simulation to complete. Why? Changing the gravityModifier affects the simulation.

- Script Update () [Scripts commonly change properties of the particle system here]

- All particle system simulations are started

- … Run other subsystems

- Script LateUpdate ()

- Rendering consumes particle simulation

If the particle simulation is still running when I call ParticleSystem.gravityModifier = 10; in LateUpdate from a script, then we might affect the simulation. There are two solutions #1 double buffer all values #2 WaitForGroup when scripts modify values that affect the simulation. Both are valid approaches. The multithreaded rendering essentially does double buffering using the command stream. Shuriken uses WaitForGroup on modify. It depends on the situation which one should be used.

For example in Shuriken we do this:

We have a global particleSystemManager.needSync. When simulation starts it is set to true. When the simulation is done it is set to false. Whenever a script modifies any values that affects the simulation eg. ParticleSystem.gravityModifier, deactivating the particle system, destroying the particle system, when Mesh data is being modified.

gParticleSystemManager.needSync is only set from the main thread. It is a very quick early out, thus we can afford to call it whenever the user changes any property. No mutex locks were used.

```
void ParticleSystem::SetEmissionRate (float rate)
{
     SyncJobs();
     m_Emissionrate = rate;
}

void ParticleSystem::BeginUpdateAll ()
{
    const float deltaTimeEpsilon = 0.0001f;
    float deltaTime = GetDeltaTime();
    if(deltaTime < deltaTimeEpsilon)
        return;


    PROFILER_AUTO(gParticleSystemProfile, NULL)


    for(int i = 0; i < gParticleSystemManager.activeEmitters.size(); i++)
    {
        ParticleSystem& system = *gParticleSystemManager.activeEmitters[i];
        if (!system.IsActive ())
        {
            AssertStringObject( "UpdateParticle system should not happen on
disabled vGO", &system);
            system.RemoveFromManager();
            continue;
        }


        Update0 (system, *system.m_ReadOnlyState, *system.m_State, deltaTime,
false);
    }


    gParticleSystemManager.needSync = true;


    JobScheduler& scheduler = GetJobScheduler();
    int activeCount = gParticleSystemManager.activeEmitters.size();
    gParticleSystemManager.jobGroup = scheduler.BeginGroup(activeCount);
```

```
    for(int i = 0; i < activeCount; i++)
    {
        ParticleSystem& system = *gParticleSystemManager.activeEmitters[i];
        system.GetThreadScratchPad().deltaTime = deltaTime;
        scheduler.SubmitJob (gParticleSystemManager.jobGroup,
ParticleSystem::UpdateFunction, &system, NULL);
    }
}


void ParticleSystem::SyncJobs()
{
    if(gParticleSystemManager.needSync)
    {
        PROFILER_BEGIN(gParticleSystemWait, NULL);
        JobScheduler& scheduler = GetJobScheduler();
        scheduler.WaitForGroup (gParticleSystemManager.jobGroup);
        PROFILER_END;


        const float deltaTimeEpsilon = 0.0001f;
        float deltaTime = GetDeltaTime();
        if(deltaTime < deltaTimeEpsilon)
            return;


        for(int i = 0; i < gParticleSystemManager.activeEmitters.size(); ++i)
        {
            ParticleSystem& system = *gParticleSystemManager.activeEmitters[i];
            ParticleSystemState& state = *system.m_State;
            system.Update2 (system, *system.m_ReadOnlyState, state, false);
        }
```

```
            gParticleSystemManager.needSync = false;
        }
    }
```

More information about Multithreaded programming:

http://www.1024cores.net/home/lock-free-algorithms/false-sharing---false

http://www.1024cores.net/home/scalable-architecture/general-recipe

Blobification

Unity now has a Blobification framework. A blob is a chunk of memory allocated in one go, it is relocatable with memcpy, and contains no virtual classes.

Our Blobification framework lets you tightly pack data, reduce duplicate data, makes assets completely relocatable, read from disk directly instead of serializing, simplify streaming, lays out memory nicely.

Overall if you write performance intensive code, especially if there is an asset which is constant at runtime, then you should really use it. In practice you can only really use it if you follow DOD.

Basic Assumption

A lot of data used in games is completely constant. For example AnimationClips and all the data of a statemachine does not change during evaluation. This is of course only true if you layout your data in a data oriented way. Meaning you separated constant data, state data that is modified on every evaluation, inputs and outputs.

Seperating what is constant and what isn't and keeping a very clear overview of what data gets modified from multiple threads is critical. The only right way to do that properly is to keep them completely seperate structs and allocate them seperately and pass them into functions that make it clear what gets modified. (rigorous const usage and not cheating)

If your data is separated by these things, it becomes possible to make assumptions and apply optimizations to the constant data that are otherwise not possible. Blobification is one of them.

What does it do?

When loading an animationclip or statemachine we don't have to deserialize anything. And since we know all data is constant, we can spend time optimizing the data at build time.

In mecanim we create structures in memory that are completely relocatable. There are plenty of pointers & arrays with different sizes in it, so you can represent complex setups. Eg. in a state machine due to the nature of what it does there are quite a few arrays and pointers referencing other states & conditions etc. States have multiple conditions which live in arrays. In a traditional model this would lead to an endless amount of allocations, and indeed when building the data it has a lot of individual allocations, but then we blobify the whole thing, we make it into a relocatable data block and a whole animationclip or statemachine can be allocated & read in one with memcpy or direclty mmapped.

Sounds cool how does it work?

All Pointers are done like this:

OffsetPtr<float> floatArray;

Instead of storing the actual pointer, it's an offset from the this pointer. It is not an indirection, it simply adds to the pointer, which in turn makes the whole thing relocatable. It is for all practical purposes as cheap as an actual pointer. And it's easy to use.

To create a blob you can allocate memory assign pointers etc.

To blobify you do this

```
struct SampleData
  {
     UInt32           floatArraySize;
     OffsetPtr<float> floatArray;
  math::float4     vec4;

     DECLARE_SERIALIZE(SampleData)
  };

  template<class T> inline
  void SampleData::Transfer(T& transfer)
  {
    TRANSFER_BLOB_ONLY(floatArraySize);
    MANUAL_ARRAY_TRANSFER2(float, floatArray, floatArraySize);
    TRANSFER(vec4);
  }

  void SetupSourceData (SampleData& data)
  {
     data.floatArraySize = 5;
     data.floatArray = new float[5];
     data.vec4 = math::float4::one;
  }

  void UseDataForStuff (SampleData& data)
  {
       ...
  }


  SampleData sourceData;
  SetupSourceData (sourceData);

  dynamic_array<UInt8> outputBlob;
  BlobWrite blobWrite (outputBlob, kNoTransferInstructionFlags,
kBuildNoTargetPlatform);
  blobWrite.Transfer(sourceData, "Base");

  // The data is now in a single block of memory, can be relocated
  // and of course can simply be casted to the SampleData or whatever data type you
were blobifying.
  UseDataForStuff(*reinterpret_cast<SampleData*> (outputBlob.begin()));
```

Take a look at BlobWriteTest.cpp for the complete code.

Internally BlobWrite uses the serialization backend. You must transfer every element and you need to serialize them in the order of the memory layout. The BlobWrite basically generates the blob, it knows about offsetPointer and can do funky stuff like the reduce copy with it. The BlobWrite also handles alignment etc correctly for you.

Magic Sauce, Reduce Copy (Massive memory reduction savings)

One of the really cool feature about blobification, is that you can use reduce copy:

```
blobWriteReduce.SetReduceCopy(true);
```

Since we know we will never modify the data in the state machine (All per instance state is kept seperate in a statememory struct) we can make any pointer to an array of conditions or other things simply reference data that already exists. Internally we simply memcmp (As long as alignment criterias are matched). In practice this reduces mecanim statemachines by around 50% in size.

For example, the particle systems in unity would be a good fit. Most data is completely constant once loaded, and I bet that 90% of all those animation curves in shuriken are exactly the same in memory, thus would give massive size savings.

Caveats

- Code must be written data oriented. You must use a very structured approach of seperated constant and state and temp data. Not a problem since you should do this anyway!
- Can't use virtual functions. Don't use that stuff...
- Can't use STL. STL should not be used in unity runtime code anyway!
- No container type support yet (Potentially it makes sense to add some simple container types, but in a lot of cases container types defeat the purpose of blobification)

But unity components have virtual functions, how do I use this thing in practice!

(Practical integration example in Mecanim)

The component system is very flexible right now, everything is modifyable at runtime. So it doesn't make sense to have blobification, between components. Essentially you need to create components or asset types, that are large enough to blobify the contents of that component or asset.

So in Unity you will have a high-level layer that uses components and virtual functions. But the actual code & data goes into a nicely data oriented C style coding blobified thing. Written with clearly seperated data for inputs / outputs / constants, explicit allocators allocating the structures. Zero virtual functions.

Essentially the OO component & asset type is just a wrapper for the data oriented low level implementation.

For example in mecanim we have an AnimationClip deriving from Unity Object, with virtual AwakeFromLoad functions, RTTI and all those OO things we learned to hate but are still useful in a lot of cases.Then we have a mecanim::animation::ClipMuscleConstant containing all the data used by mecanim. The AnimationClip is simply responsible for making sure that constant is available and can be fetched and integrates nicely with editing tools which use a a completely different data structure for editing the clips.

The Animator component is responsible for collecting all associated animationclips, find the statemachine in the AnimatorController (StateMachineConstant), ensuring that all animationclips are loaded and if everything is in order, the workspace memory is created. Something like this. Basically this part is responsible for creating all the low level things. They only succeed if all data is valid. If anything is not valid, we never even create the low level structures and the High level API at runtime simply checks for existance of the m_ControllerConstant. And if it exists it knows EVERYTHING is valid.

This approach also simplifies the code a lot because in the low level layer, there is no need for any kind of error checking at all, since we know everything is validated at the higher level in a single place.

```
class Animator : Component
{
    virtual void AwakeFromLoad ();

    // Animator is a virtual component. The implementation lives in the mecanim
folder and is purely c-style data oriented code.
    mecanim::animation::AvatarWorkspace*        m_AvatarWorkspace;
    mecanim::animation::ControllerMemory*        m_ControllerMemory;
    mecanim::animation::AnimationSet*            m_AnimationSet;



void Animator::CreateObject()
{
 ClearObject();
 // Avatar object creation and initialization related
 if (!m_Avatar.IsValid())
  return;
 mecanim::animation::AvatarConstant* avatar = m_Avatar->GetAsset();
 if (avatar == NULL)
  return;

 if (!BindSkeleton())
  return;

 m_AvatarConstant = avatar;
 m_AvatarMemory = mecanim::animation::CreateAvatarMemory(avatar, mAlloc);
 m_AvatarInput = mecanim::animation::CreateAvatarInput(avatar, mAlloc);
 m_AvatarWorkspace = mecanim::animation::CreateAvatarWorkspace(avatar, mAlloc);
 m_AvatarOutput = mecanim::animation::CreateAvatarOutput(avatar, mAlloc);


 // Controller object creation and initialization related
 AnimatorController* animatorController = m_Controller;
 if (animatorController == NULL)
  return;

 m_ControllerConstant = animatorController->GetAsset();
 if (m_ControllerConstant == NULL)
  return;

 AnimationClipVector& clips = animatorController->GetAnimationClips();
 for (AnimationClipVector::iterator it = clips.begin(); it != clips.end(); it++)
 {
  AnimationClip* clip = *it;
  if(clip && clip->GetRuntimeAsset() == NULL)
  {
   WarningStringObject (Format("Animatorcontroller failed to load AnimationClip
%s",clip->GetName()), this);
  }
 }

 animatorController->AddObjectUser(m_AnimatorControllerNode);

 m_AvatarWorkspace->m_ControllerWorkspace =
mecanim::animation::CreateControllerWorkspace(m_ControllerConstant,
m_AvatarConstant, mAlloc);
```

See also streamedclip.cpp and StreamedClipBuilder.cpp.

Multithreaded Rendering

Unity supports multithreaded rendering on desktop platforms, Xbox 360 and possibly mobile in the future.

It works by taking advantage of the abstract GfxDevice interface that the different renderers inherit from. When threading is enabled, there is a GfxDeviceClient instance that rendering code uses on the main thread. The client forwards all the rendering commands to a worker (GfxDeviceWorker) that listens on another thread. The worker owns the real graphics device that does the actual rendering.

The two graphics devices in this setup are accessed through GetGfxDevice() and GetRealGfxDevice() functions, which check that you call them from the correct thread. The real device must only be used from the rendering thread, except in rare cases where the main Unity thread takes ownership of the device.

Ring buffer

The communication between the main and render thread happens through a ThreadedStreamBuffer, a single-producer, single-consumer ring buffer. This will automatically block when one thread waits on the other, either because there is no data or the buffer is full. It's recommended to use this for other things than rendering, since it handles the issue with idling gracefully (with semaphores) when it's blocked. When it's not blocked there is very little synchronization cost. Notice you must explicitly submit what you write before it can be read by the consumer thread, with WriteSubmitData(). Equally you must release what you've read, with ReadReleaseData().

Display lists

The GfxDeviceClient class supports a special feature, which is recording a list of rendering commands and playing them back later. This allows optimizing the rendering code by caching the output between frames, when we know nothing has changed. It's currently used by materials applying their passes but could be used for other things. It both reduces the time taken by the main thread and the bandwidth needed to send to the render thread, since the "display lists" are stored outside the ring buffer. There is a system for patching any global properties used in the rendering commands, since they can change without the material knowing about it. Those properties are sent through the ring buffer every time the display list is run.

Recording display lists can fail either if the device doesn't support it (non-multithreaded GfxDevices) or if something happened during recording that means we don't want to play back the same commands again. For example, uploading a texture during recording would cause the same texture to be uploaded every frame from now on, which would be very inefficient. If that happens, EndRecording() will return failure and the display list will not be reused.

Debugging

For debugging it's often useful to disable threaded rendering, since it makes it harder to see where the GfxDevice calls originated from. You can disable it on desktop by running with the option `-force-gfx-direct`. If you need the client device enabled, for example to use display lists, choose `-force-gfx-st` instead. It's also possible to run multithreaded in lock-step mode, where the main thread waits for each command on the render thread to finish. For lock-step mode you need to recompile the source after changing `DEBUG_GFXDEVICE_LOCKSTEP` in Runtime/GfxDevice/threaded/GfxDeviceWorker.h to `1`.

GfxThreadableDevice functions

Mostly the extra virtual functions in the GfxThreadableDevice class are variations of GfxDevice functions that took data that wasn't thread-safe, for example ShaderLab::PropertySheet. When calling SetShaders() in multithreaded rendering, the main thread takes a property sheet and turns it into plain serialized data that can be fed to SetShadersThreadable() on the render thread. The function to implement when adding a new platform is the threadable version, and the feeding of the data is handled by cross-platform code.

CreateShaderParameters() exists because the main thread needs to know which parameters a shader takes, and on GL-like platforms it can vary with different fog modes because we recompile shaders if the mode was changed. Instead of compiling everything up front, we stall the main thread while the render thread figures out what the shader needs, when a shader is used with a fog mode that wasn't used before.

IsCombineModeSupported() exists because we want CreateTextureCombiners() failure to happen on the main thread, not the render thread where we can't do anything about it. If it returns true creating combiners is supposed to succeed.

Editor

How to serialize stuff (C#)

General rules about serializing data

1. Always aim to serialize the minimal set of data. The primary purpose of this is not even saving disk space, but making sure that maintaing backwards compatibility does not become a huge pain 1 month later.
2. Never serialize duplicate data or cached data. This is a huge hassle for backwards compatibility and generally an annoyance because you can almost safely assume that data will find a way to get out of sync. If you have two pieces of data that are serialized and represent the same value, you are in trouble. So just never ever serialize caches, unless there is a really really really good reason and there is a really really solid framework built around the functionality
3. Try to serialize data that is as directly modified by the user as possible. It's fine to generate data on the fly from the serialized data.

What does Unity Serialize

Requirements for serialization on the type that is serialized

1. Several builtin types are supported for serialization (int, float, string, bool, Vector3, ...)

2. Classes are serialized if they are marked with the System.Serializable attribute
3. List and array types are serialized if the class or builtin type that it contains is supported. Other container types are not supported.
4. Only member fields are serialized, static fields or properties are never serialized
5. Structs (unless they are supported builtin types) are never serialized
6. Since the layout of a serialized structure always needs to be the same independent of the data and only dependent on what is exposed in the script. You can not have any nested, recursive structures where you reference other classes. The only way to have references to other objects is through UnityEngine.Object derived classes. These classes are completely seperate and only reference each other but dont embed the contents.

What variables are serialized

1. public variables are always serialized
2. Private variables can be serialized using the [SerializeField] attribute

Making editor code fully reloadable

When reloading scripts, Unity stores all variables in all loaded scripts, and after loading the scripts restores them.

When reloading scripts, unity restores all variables - including private variables - that fulfill the requirements for serialization, even if it has no SerializeField attribute. In some cases you specifically need to prevent private variables from being restored, eg. you want a reference to be null after reloading from scripts. In that case use the [System.NonSerialized] attribute.

Static variables are never restored. So don't use static variables for state that needs to be retained over script reloads.

Style Guide

This is not a complete style guide at this point, but contains some important information about how to work with the EditorGUI styles to make them consistent.

Correct margin and padding for common GUI styles

Property	left,right,top,bottom
margin	4,4,2,2
padding - label based styles	2,2,1,2
padding - text field based styles	3,3,1,2
padding - minibutton/popup based styles (uses mini font)	6,6,2,2

Colors and effects for GUI styles

Keyboard focus colors and Photoshop effects

Text color on keyboard focus:

rgb 0, 50, 230 (light skin)

rgb 76, 126, 255 (dark skin)

Control effect on keyboard focus:

Color - used for stroke, flow, overlay - same for light and dark skin: rgb 51, 87, 217

Inner stroke: size 1, position inside, blend mode normal, opacity 100%

Inner glow: blend mode normal, opacity 42%, technique softer, source edge, choke 100%, size 2, range 50%

Color overlay (to make objects match selection color, like color picker icon): Hard light, opacity 100

Attempted overview over what different styles are used for

Was different for LookLikeInspector and LookLikeControls

Style in EditorStyles	LookLikeInspector version	LookLikeControls version
m_Label	IN Label	ControlLabel
m_Popup	IN Popup	MiniPopup
m_TextField	IN TextField	TextField
m_ColorField	IN ColorField	ColorField

m_ObjectField	IN ObjectField	ObjectField
m_ObjectFieldThumb	IN ObjectField	ObjectFieldThumb
m_Toggle	IN Toggle	Toggle
m_Foldout	IN Foldout	Foldout
m_FoldoutSelected	IN SelectedLine	none

Was shared between LookLikeInspector and LookLikeControls

- IN BigTitle
- IN Title
- IN TitleText

Used elsewhere

Style name	Used for
IN BigTitle Inner	used in Editor.cs
IN DropDown	used in AnnimationWindow.cs and ParticleSystemStyles.cs
IN LockButton	used in ObjectBrowser.cs, InspectorWindow.cs, ParticleSystemWindow.cs
IN ThumbnailSelection	used in PlayerSettingsEditor.cs
IN ThumbnailShadow	used in PlayerSettingsEditor.cs and ParticleSystemStyles.cs

Not used?

- IN FoldoutStatic

Working with Editor Styles

The styles that are used in the editor itself live in editor_resources. See the page Editor Resources for details.

Rather than authoring these styles in the form of GUISkins (as we've done previously) we now have EditorStyles.

What are EditorStyles?

An EditorStyle resembles a GUIStyle in that it has all the same settings. However, EditorStyles support inheritance, and the light and dark version of a style is contained in the same asset.

> EditorStyles are only used in the editor_resources project. They are compiled into regular GUISkins that don't have inheritance. They are only a workflow improvement for people who create and work with these styles. They don't change the Editor itself in any way, since they don't exist in the Editor.

Features of EditorStyles

- EditorStyles support **inheritance** so one EditorStyle can inherit from another and only override selected properties. This makes it easier to manage related styles and ensure consistency.
- The **same asset** contains both the **light and dark skin** version of a style. (The data model supports extending with more skins if we want.) The light skin version is the default, and the dark skin version specifies how it differs from the light skin version.
- An EditorStyle can specify that it's **shared**, meaning that the light and dark skin versions are identical.
- An EditorStyle can specify that it's **abstract**, meaning that it can be used as a common base style for other styles, but won't be included in the compiled skins itself.

Overview of tools and workflow

Project View Structure

All EditorStyles are located in a folder called EditorStyles. They are further divided into categories using folders.

Editor Style Window

The Editor Style Window can be opened with the menu: `Tools > Edit Styles...`

So far, the window is used to provide a better overview of the styles, though more functionality will be added later.

In the left side, the categories are shown, which corresponds to folders inside the EditorStyles folder.

In the right side, the styles in that folder are shown according to their inheritance hierarchy. All parent styles are included in this hierarchy, even if they are not in the same folder.

Clicking on a style selects the asset and shows it in the Inspector.

Editor Style Inspector

The Inspector for an EditorStyle is divided into three parts:

- Top part with general settings (Is Abstract setting, Parent field, Shared setting, toolbar to select skin version).
- Scrollview with the actual style properties. The shown properties are for the selected skin version (base or dark).
- Bottom part with tools (Bake All, Clean Redundant).

Reading the property inheritance

All properties have a toggle to the right that indicates if this property is overridden in this style.

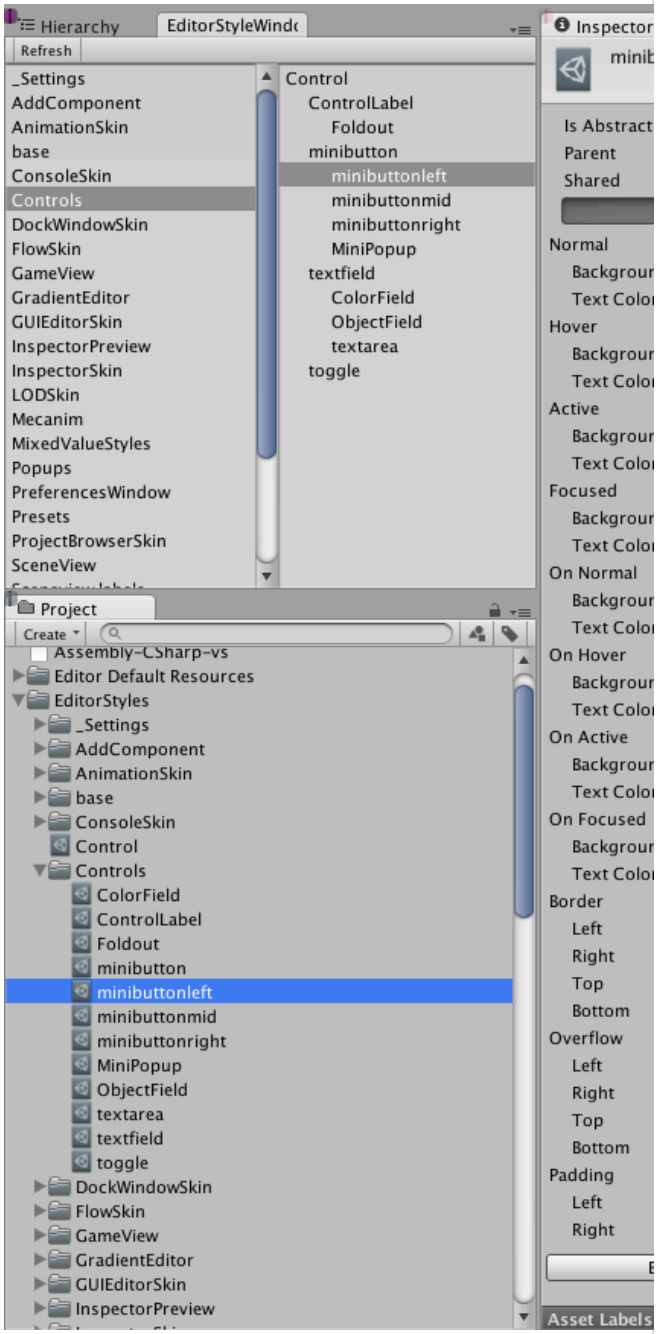
If an overridden style property is different from the inherited value, the inherited value is shown to the right of the toggle for comparison.

If an override toggle is enabled but there is nothing to the right of it, it means that the overridden value is the same as the inherited value.

Tools

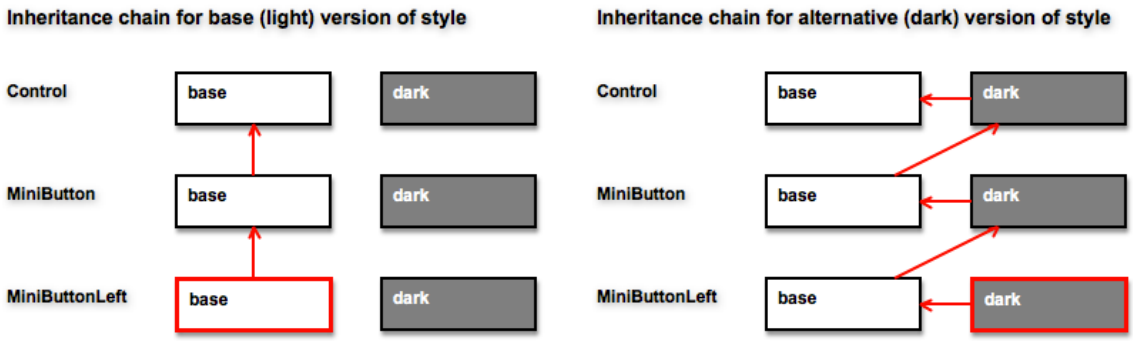
The *Bake All* button makes all properties overridden. Those that were not already overridden will use the inherited value and override with that same value. This tool does the same as if you enable all the override toggles.

The *Clean Redundant* button removes override data for all properties that have the same value as the inherited value. This tool does the same as if you disable all toggles that don't show a value to the right.



Inheritance for light and dark versions

For the base version of a style (the light skin version), property inheritance works in a straightforward way. If the (light) style doesn't define an override value for the property itself, we look at the (light) parent. If that one doesn't define an override value either, we look at that one's (light) parent, and so on.



For the alternative version of a style (the dark skin version), property inheritance is a bit more complicated, but there's a good reason for it, and it works in a fairly intuitive way once understood.

The a dark skin version of a style, we want inheritance to still work: The dark version of a style should inherit from the dark version of the parent style. But at the same time, we *also* want the light and dark versions of a style to share as much data as possible, so the work of keeping them consistent is as easy as possible. This is implemented by making the dark skin version inherit the values from the light skin version.

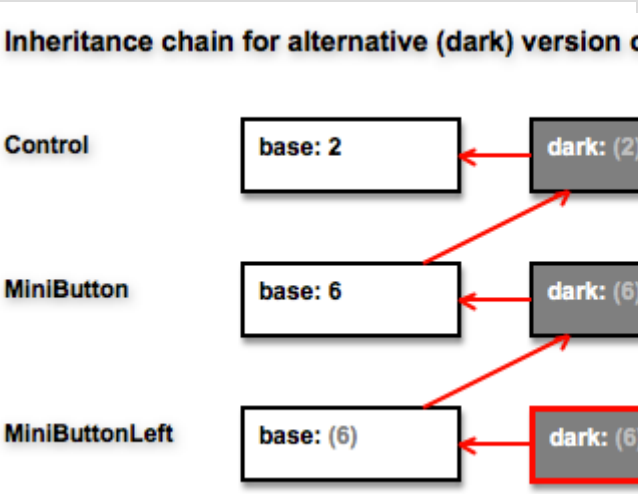
How can the dark version have two parents?
How can the dark skin version of a style inherit both from its own light skin version, and from the dark skin version of the parent? We do this by looking at first the style itself, then the parent style, and then the parent's parent style and so on, similar to how it works for the light version. But for each of these steps, we look first at the dark skin version, and if that doesn't define an override value, then we look at the light skin version. The chain is illustrated in the figure above.

Why this approach?
The reason for this inheritance chain is that it does what we need. Let's look at a few use cases.

Use case: A property that's the same for the light and dark skin

Most properties of styles should be the same for the light and dark skin; for example the margin values should typically be the same. We want to only specify the margins in the light skin versions of the styles and let the dark skin versions use the same values.

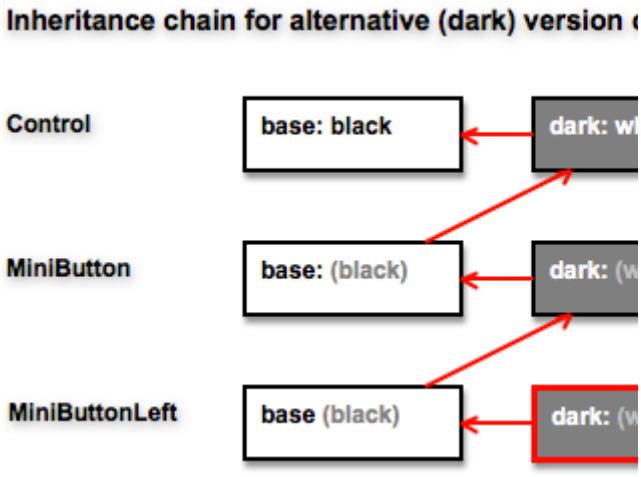
In the illustration to the right, the light version of the Control style specifies a margin of 2, while the light version of the MiniButton style overrides with a margin of 6. All the values shown in parenthesis are inherited from these according to the inheritance chain.



Use case: A property that's different for the light and dark skin

Some properties of styles should be different for the light and dark skins; for example text color should typically be different. We want to be able to specify this difference in a parent style, and if a child style doesn't override the values, it should inherit this difference.

In the illustration to the right, the light version of the Control style specifies a text color that's black, while the dark version of the Control style overrides with a text color that's white. All the values shown in parenthesis are inherited from these according to the inheritance chain. Note that when walking the chain, only actual overridden values are picked, while inherited values are ignored. For example the dark version of the MiniButton style doesn't inherit the value of black from the light version of the MiniButton style, because that value is only inherited. Instead it goes on to look at the dark version of the Control style and pick the value there since it's overridden there; not inherited.



Writing Custom Inspectors

Guidelines for how to write a custom Inspector/Editor. The way to do it has changed after we're introduced multi-object-editing. Here's the guidelines.

Add the CanEditMultipleObjects attribute to the editor class

```
[CustomEditor (typeof (Camera))]
[CanEditMultipleObjects]
internal class CameraEditor : Editor
{
 ...
```

Everything should be based on SerializedProperties whenever possible

Editors should preferably support:

- Multi-object-editing - showing if all selected objects have the same value for certain fields or not, and applying a value to them all if modifying it.
- Showing if the value on a prefab or model instance is overridden (shown in bold) or not, plus having the Revert to Prefab menu.

This means using SerializedProperties and the corresponding GUIControls that handle them. The Editor class has a serializedObject property that should be used for all SerializedProperties.

```
SerializedProperty m_ClearFlags;
SerializedProperty m_BackgroundColor;
SerializedProperty m_NormalizedViewPortRect;
SerializedProperty m_FieldOfView;
SerializedProperty m_Orthographic;

public void OnEnable()
{
 m_ClearFlags = serializedObject.FindProperty ("m_ClearFlags");
 m_BackgroundColor = serializedObject.FindProperty ("m_BackGroundColor");
 m_NormalizedViewPortRect = serializedObject.FindProperty ("m_NormalizedViewPortRect");
 m_FieldOfView = serializedObject.FindProperty ("field of view");
 m_Orthographic = serializedObject.FindProperty ("orthographic");
}
```

Anything that can should just use PropertyField.

```
EditorGUILayout.PropertyField (m_ClearFlags);
```

If a value should be represented with a control that's not the standard for that property type, use a GUI control overload that takes a SerializedProperty if possible. For example, using PropertyField on a float property will just create a FloatField. If a slider is desired instead, use the Slider overload that takes a SerializedProperty.

```
EditorGUILayout.Slider (m_FieldOfView, 1f, 179f, new GUIContent ("Field of View"));
```

If more control is needed over the assigned value - for example limiting it to within a certain range - the value can be re-assigned manually if it was modified by a control:

```
EditorGUI.BeginChangeCheck ();
EditorGUILayout.PropertyField (m_PositiveValue);
if (EditorGUI.EndChangeCheck ())
 if (m_PositiveValue.floatValue < 0)
  m_PositiveValue.floatValue = 0;
```

What do do when SerializedProperty overloads don't cut it.

Sometimes it's not possible to pass a SerializedProperty directly to a GUI control. For example, for the Camera editor m_Orthographic is a bool but we want to show a popup instead of a toggle to make the choices more explicit. There's no GUI controls that show a popup based on a toggle.

Instead, 2 things can be done instead:

1: Show that the values are different (EditorGUI.showMixedValue)

Regular GUI controls that don't take a SerializedProperty can be told to show a dash or other indication that the values are different for the selected objects by setting EditorGUI.showMixedValue to true while the GUI control function is called. It should be set to true if the values *are* different. The easiest way to determine that is with the property on the SerializedProperty called hasMultipleDifferentValues.

```
EditorGUI.showMixedValue = m_Orthographic.hasMultipleDifferentValues;
ProjectionType projectionType = (ProjectionType)EditorGUILayout.EnumPopup("Projection",
oldProjectionType);
EditorGUI.showMixedValue = false;
```

2: Apply the *changed* value to all the objects

When the user has changed the value, it should be applied to all the objects. Note that this should ONLY be done when the value has ACTUALLY BEEN CHANGED.

- **DO NOT check if the new value is different than the old.**

When multiple objects are selected the "old value" is meaningless since it's only the old value of ONE of the objects (at random). Consider two objects with radius 2 and 3. The user selects both and see a dash meaning the values are different. He types in 3 and hits enter. However, the "old value" happens to be 3 as well, so comparing the new value with the old show that they're the same and as a result no value is changed -> BUG.

- **DO check if GUI.changed was set to true inside the function.**

All GUI controls are now required to strictly set GUI.changed to true if and only if the user changed the value.There is a convenience pair of functions to cehck for GUI.changed.

```
EditorGUI.BeginChangeCheck ();
// GUI control here...
if (EditorGUI.EndChangeCheck ())
 // Assign value to all objects here
```

Actually assigning the value back to the objects is easiest to do by just setting the value of the SerializedProperty if possible. If it's not the value can be assigned manually to all the objects in the targets array but this should be a last resort.

Writing GUI Controls

Guide to writing Editor GUI Controls like the ones in EditorGUI and EditorGUILayout classes.

Overloads

Most GUI controls have many overloads. Here are the de facto guidelines for which overloads should be implemented for a control.

If a GUI control is to ever be used in an inspector (an Editor) it should preferably support:

- Multi-object-editing - showing if all selected objects have the same value for this field or not, and applying it to them all if modifying it.
- Showing if the value on a prefab or model instance is overridden (shown in bold) or not, plus having the Revert to Prefab menu.

Both of these means that the control should have overloads that take a SerializedProperty.

Variations of Vanilla Overloads (that just take and return a value)

Its most common for vanilla overloads to come in 6 variations. There are overloads for no label, string label, and GUIContent label, and for each of those an extra overload that takes a GUIStyle to use for the control:

- (value)
- (value, GUIStyle)
- (string label, value)
- (string label, value, GUIStyle)
- (GUIContent label, value)
- (GUIContent label, value, GUIStyle)

The order of parameters is:

- Rect (for EditorGUI versions)
- Label (when present)
- Value
- Other parameters (when present)
- GUIStyle (when present)
- params GUILayoutOptions[] (for EditorGUILayout versions)

Some controls don't support any GUIStyle parameters. Current practice shows that:

CAN override GUIStyle:

- Controls based on a *single* text field (TextField, IntField, PasswordField, etc.)
- Controls based on a popup or dropdown (Popup, LayerField, TagField, etc.)
- Label, SelectableLabel, Foldout

CANNOT override GUIStyle

- Controls based on a *multiple* text fields (Vector3Field, RectField, BoundsField, etc.)
- All other GUI controls not mentioned.

Variations of SerializedProperty Overloads

It's most common for SerializedProperty overloads to come in 2 variations:

- (SerializedProperty)
  One that does not take a label parameter and instead uses the displayValue of the property for its label.
- (SerializedProperty, GUIContent label)
  One with an explicit GUIContent label. Note that if the label is GUIContent.none, no label should be displayed, not even layout spacing for it.

The order of parameters is (note that the label is *last* contrary to vanilla overloads):

- Rect (for EditorGUI versions)
- SerializedProperty
- Other parameters (when present)
- Label (when present)
- params GUILayoutOptions[] (for EditorGUILayout versions)

Special for labels in SerializedProperty overloads

SerializedProperty overloads should not have overloads that take a string label (only current exceptions are Slider and IntSlider), and they never take a GUIStyle.

The convention that the label can be removed by specifying GUIContent.none was established since all existing controls interpreted no label parameter as meaning that the label should be based on the property name and there was no way to not have a label. Note that if the control uses the PrefixLabel methods for handling the label, it will automatically be removed for GUIContent.none with no extra work required.

Invoking GUI controls through PropertyField

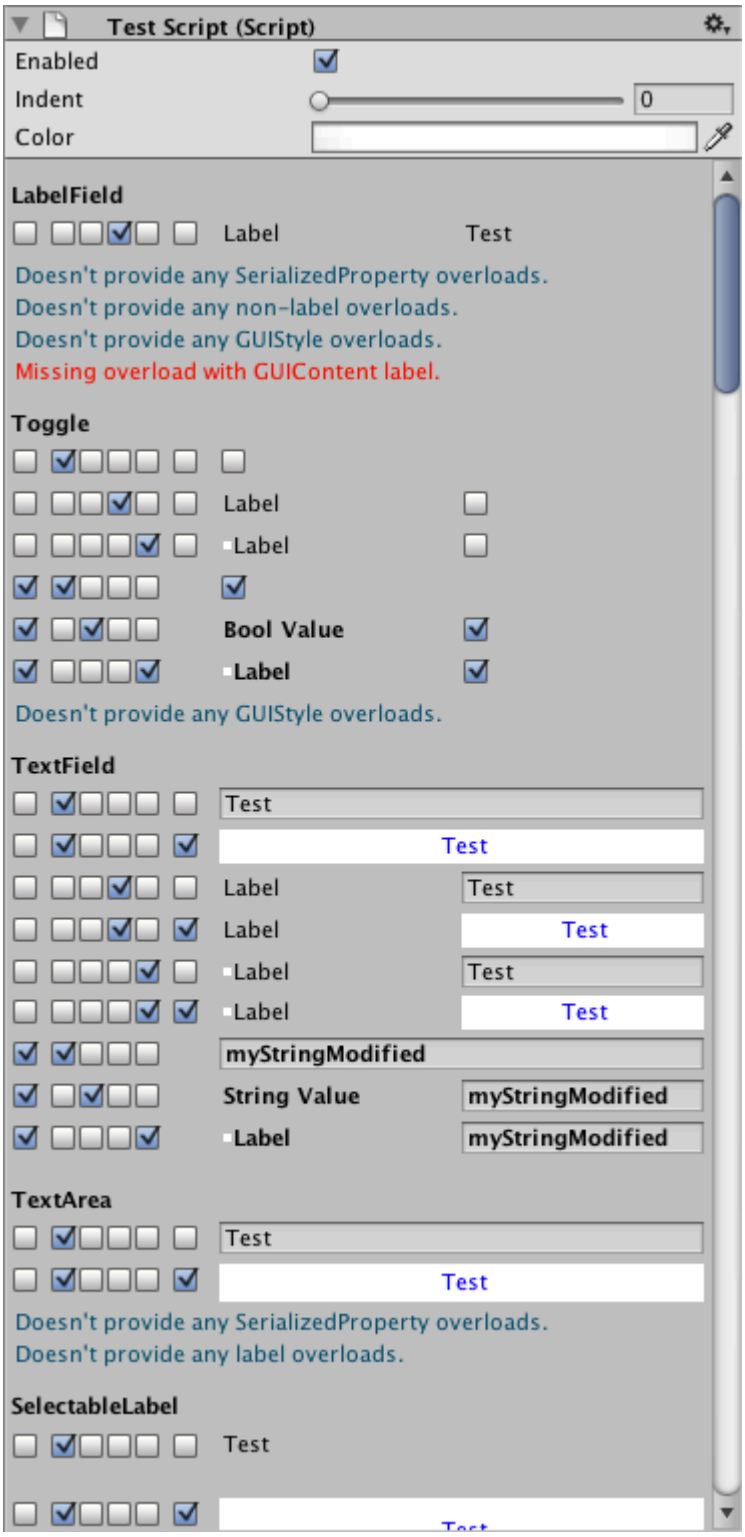
If a GUI control is the "default" representation of a certain data type, the control should not be invoked directly but instead be invoked through the PropertyField. Make the PropertyField use the control and make the control private so it can't be called directly. For example, the Vector3Field that takes a SerializedProperty is private and is called by PropertyField for Vector3 types.

Testing

We have a manual GUI verification framework to do "assisted manual verification" that GUI controls look and work as they should and have all the right overloads. It's a project folder located in the source checkout in:

`Testing/ManualGUIVerification`

- Open the project folder.
- Open the scene GUIVerification inside it.
- Select the prefab instance in the scene.
- The testing framework is shown in the Inspector.



Adding a new control or overload to the framework

When you have made a new EditorGUI control or changed an existing one, make sure it's properly tested in the framework:

- Open the script `TestScriptEditor`
- Make sure the GUI control is tested inside the TestingGUI function with all the overloads.

Example:

```
BeginControl ("Slider");

DoPermutation ();
m_Float = EditorGUILayout.Slider (m_Float, kSliderLeft, kSliderRight, layoutOptions);
DoPermutation (Param.LabelString);
m_Float = EditorGUILayout.Slider (labelStr, m_Float, kSliderLeft, kSliderRight, layoutOptions);
DoPermutation (Param.LabelContent);
m_Float = EditorGUILayout.Slider (label, m_Float, kSliderLeft, kSliderRight, layoutOptions);

DoPermutation (Param.Property, Param.LabelNone);
EditorGUILayout.Slider (m_FloatProp, kSliderLeft, kSliderRight, noLabel, layoutOptions);
DoPermutation (Param.Property);
EditorGUILayout.Slider (m_FloatProp, kSliderLeft, kSliderRight, layoutOptions);
DoPermutation (Param.Property, Param.LabelString);
EditorGUILayout.Slider (m_FloatProp, kSliderLeft, kSliderRight, labelStr, layoutOptions);
DoPermutation (Param.Property, Param.LabelContent);
EditorGUILayout.Slider (m_FloatProp, kSliderLeft, kSliderRight, label, layoutOptions);

EndControl ();
```

Each GUI control has a block wrapped in BeginControl and EndControl. Inside the block is each function overload is listed with a call to DoPermutation above specifying what type of standard parameters are used in this overload.

- For vanilla overloads, pass one of the existing variables if possible (m_Float, m_Color, etc.) or add a new variable for a new type at the top of the file if needed.
- For SerializedProperty overloads, pass one of the existing SerializedProperty variables if possible (m_FloatProp, m_ColorProp, etc.) or add a new one if needed.
    - Add the variable of the right type to TestScript.
    - Add the SerializedProperty variable at the top of the TestScriptEditor file.
    - Do FindProperty on the SerializedProperty inside OnEnable.
- If a control has a SerializedProperty version that does through PropertyField, just use the function `IncludePropertyFields` instead of writing the individual function calls manually.

Example of control that uses PropertyField for its SerializedProperty variants:

```
BeginControl ("FloatField");

DoPermutation ();
m_Float = EditorGUILayout.FloatField (m_Float, layoutOptions);
DoPermutation (Param.Style);
m_Float = EditorGUILayout.FloatField (m_Float, style, layoutOptions);
DoPermutation (Param.LabelString);
m_Float = EditorGUILayout.FloatField (labelStr, m_Float, layoutOptions);
DoPermutation (Param.LabelString, Param.Style);
m_Float = EditorGUILayout.FloatField (labelStr, m_Float, style, layoutOptions);
DoPermutation (Param.LabelContent);
m_Float = EditorGUILayout.FloatField (label, m_Float, layoutOptions);
DoPermutation (Param.LabelContent, Param.Style);
m_Float = EditorGUILayout.FloatField (label, m_Float, style, layoutOptions);

IncludePropertyFields (m_FloatProp);

EndControl ();
```

Testing that a control works correctly

When a control with all its overloads is added to the framework it's easy to see at a glance:

- If it looks right.
- If it respects indentation correctly (use the indent slider at the top).
- If it respects GUI.color and GUI.enabled correctly (use controls at the top).
- If overloads that take a SerializedProperty correctly are shown in bold for modified values on a prefab instance, and have the Revert right click menu.
- If the various overloads behave correctly.
    - Does all the overloads that take a GUIStyle actually use the style?
    - Does all the overloads that take a label actually use the label? Including the image, for GUIContent labels?
    - Do overloads that don't take a label indent correctly and allocate the right space if it's a multi-line control?

The framework will also display information about what overloads are provided which can help to decide if some overloads are missing, or if some are present which shouldn't be there.

- Red text indicates gross inconsistency in which overloads are provided and this should be always fixed.
- Other text indicates presence or absence of certain overloads which may or may not be intentional. See rules further up the page for deciding which overloads should be available.

Known Issues

- The GUI Verification Framework breaks for some reason when opening a popup window (like ObjectField, ColorField, and

CurveField does) due to mismatched layout.
- The GUI Verification Framework currently only tests EditorGUILayout overloads, so it doesn't yet test consistency of the EditorGUI overloads (there's not always 1:1 in which overloads are provided).

Documentation

Overloads are grouped together in the docs by using a listonly syntax above each function except the last one: `/// *listonly*`

When a GUI control have both vanilla and SerializedProperty overloads, the vanilla ones are grouped together and the SerializedProperty ones are grouped together:

```
/// *listonly*
CSRAW vanilla function here...

/// *listonly*
CSRAW vanilla function here...

/// Actual docs with description and params here
CSRAW vanilla function here...

/// *listonly*
CSRAW SerializedProperty function here...

/// Actual docs with description and params here
CSRAW SerializedProperty function here...
```

The functions should appear in the txt file in the order described under Overloads. They will appear in the same order in the docs as they do in the txt files.

Tips&Tricks

Tips & Tricks for Unity Developers

This page gathers an assortment of tips&trips for working with and on Unity source code.

Debugging Techniques

Problem	What might be happening	What to do
Player complains about corrupt data	There's probably trouble in the serialization code causing the editor to serialize data differently from what the player expects.	1. Go to BuildPlayerUtility.cpp and set DEBUG_FORCE_ALWAYS_WRITE_TYPETREES to 1. 2. Go to SerializedFile.cpp and set "#if 0 // Log when loaded typetree is out of sync." in SerializedFile::ReadObject to 1. 3. Build editor and player. 4. Run editor and build player from editor. 5. Launch player and check the player log for typetree mismatches reported by the serialization system.
Editor rejects Asset Bundle as having an incompatible runtime version but you need the bundle to load to reproduce a bug.		1. Go to AssetBundleUtility.cpp and in TestAssetBundleCompatibility() disable the runtime version check. ATM, this is safe to do in the editor. In players, it can lead to broken shader references.

Binary2Text and WebExtract Shortcuts (Windows only)

To add right-click shortcuts to Binary2Text and WebExtract to the Windows Explorer, modify the attached file to point to valid binary2text.exe and WebExtract.exe (found in the Data/Tools folder of every Unity editor build) executables on your local harddrive and then simply double-click the file and confirm.

This allows you to dump binary Unity files to text or unpack asset bundles through a simple right click.