Resolution Games Style Guide

These are guidelines to keep a Unity project tidy and easy to work on.

Version Control

Configure Git

Make sure you configure Git to avoid merge commits when you pull. In short, before you clone any project, make the setting:

Command line:

```
git config --global branch.autosetuprebase always
```

SourceTree:

Options → Git → Use rebase instead of merge for tracked branches

If you have already cloned, it may be too late to fix it this way. See the link above.

Using Version Control

Commit one change at a time. I.e. don't make changes to several different "things" (e.g. make some graphical edits to the scene as well as changing an unrelated script) and create a single commit out of it. `git add -p` is your friend here. Rationale: This makes it easier to merge, cherry-pick and resolve conflicts, and makes it easier to find when a bug was introduced (e.g. with `git bisect`).

When trying out things, do the changes in a temporary scene. When you are sure what you want to commit, make the correct changes in the main scene. This is to avoid committing changes that are not related to what you actually wanted to change. Alternatively, make experimental changes in the scene, then revert all of them before doing the final changes.

Every asset in Unity has a corresponding meta file. Unity makes sure that they are in sync with the assets. Unity only does this when it's the active application. Always switch over to Unity before committing to make sure that any meta files that don't have a matching asset are removed, and that all added assets have a corresponding meta file. Always commit the meta files together with the assets.

Git can't handle empty directories. Remove empty directories to make sure Unity deletes its corresponding meta file. Materials directories tend to be empty. Remove those.

After upgrading Unity, or importing assets made with an older version of Unity, the assets or its meta files may show changes even if you didn't change anything. If this keeps happening, make a separate commit with only these "Unity enforced changes" to avoid getting them in the future. This will also help everyone else.

Assets

For new projects, make sure the Assets directory is directly under the project's root. This is to avoid too long paths. Also, the build system is designed for this.

Put all assets under a folder called RG under Assets. Use subfolders as necessary. This is so we know which files we made and which are made by other people. Among other things, we can use this to ignore compiler warnings in code we didn't write.

"Assets/Standard Assets" is only for third party assets. Do not put our own assets there. Do not edit the assets here.

Put "art", i.e. textures, materials, meshes etc. in a folder called Art below Assets/RG. Use subfolders to split assets into logical groups.

Put script assets into other folders depending on what area of the game they are about, e.g. "Steering", "Environment", etc.

Remove assets that are no longer in use.

When updating an asset, just replace the old asset. Do not add the new asset with a new name and update all references to the old one with the new one.

Use PascalCase for naming assets (every word starts with a capital letter). Do not add underscores or spaces between words.

IF YOU'RE NOT A PROGRAMMER, YOU CAN STOP READING HERE

Code Style

Whitespace

Put spaces around operators like =, ==, <, >, +, -, etc. Put space after comma. Do not put space before or after parenthesis in function calls or if/while/for statements. Put whitespace around {}.

Example:

```
// DON'T:
if (x+1==Foo(1,3) ){
    SomeFunction ();
}

// DO:
if(x + 1 == Foo(1, 3)) {
    SomeFunction();
}
```

Indent with spaces. Rationale: It makes output consistent in all tools, e.g. diff, git show, etc. It simplifies copy/paste from the terminal. Its WYSIWYG nature makes it harder to make mistakes.

Use 4 spaces for indentation.

Do not keep spaces at end of lines. Rationale: This can give false diffs.

End text files with a single line break. Do not have blank lines at the end of the file. Rationale: This avoids false diffs and avoids Git complaining about missing line break at end of file.

Use Unix line endings (LF only), not Windows (CR+LF). TODO: Figure out how to set this up in Git and Unity for Windows so it *just works*. Visual Studio is a problem here, as it has no easy way to work with Unix line endings.

Braces

Put the opening brace on the same line as its opening statement (e.g. *if* or *while*). Rationale: This saves vertical space and keeps lines that belong together together. It is still easy to see which closing brace aligns with the starting brace, as the lines will have the same indentation. Example:

```
public void UpdatePositions() {
    while(!done) {
        if(foo < 0) {
            done = true;
        }
    }
}
```

Always add braces to `if`, even if there is only one statement. Rationale: This avoids unnecessary diffs or bugs when you add statements.

```
// DON'T:
if(x == 0)
    Bar();


// DO:
if(x == 0) {
    Bar();
}
```

This rule also applies to `for` and `while` statements.

Put `else` on a separate line:

```
// DON'T:
if(x == 0) {
    Foo();
} else {
    Bar();
}


// DO:
if(x == 0) {
    Foo();
}
else {
    Bar();
}
```

Explicit access modifier

Class fields are private by default, but add the private keyword anyway. Rationale: Explicit is better than implicit. It's hard to tell whether it was supposed to be private or if it was a mistake.

Exception: The methods with "magic names" like Update or OnDestroy that Unity finds no matter what their access modifier says. As the access doesn't matter, leave it out.

```
// DON'T:
public class X {
    int someValue;

    void SomeFunction() {}

    void Update() {}
}

// DO:
public class X {
    private int someValue;

    private void SomeFunction() {}

    void Update() {} // exception to the rule
}
```

Naming Conventions

Use American English for all names. Use names that clearly explains whatever it names, without it getting overly long. For variables of small scope (e.g. a local variable inside a short function), single letter variables are allowed if it makes the code clearer. Well, actually, anything that makes the code clearer is allowed. :-) (Though remember that it may be only in your subjective opinion, so check with others.)

Common errors: Wrong capitalization on properties (you know, those things with get/set keywords, should be upper case) and instance variables (should be lower case, no matter if they are public or private).

Kind	**Naming convention and example**

Class	Noun phrase in PascalCase:
	class Fish {}
	class AnalyticsReport {}

Method	Verb phrase in PascalCase:
	void UpdatePosition() {}
	bool IsAlive() {}

Property (with get/set keywords)	Noun phrase in PascalCase:
	string Name { get; set; }
	int NumberOfParticles { get; }

Variable (member, argument, local, …)	Noun phrase in lowerCamelCase:
	Vector3 position;
	float distanceToTarget;
	GameObject enemy;

Enumeration	Noun phrase in PascalCase:
	enum Color { Red, White, Yellow }
	enum Direction { Up, Down }

Enumeration member	PascalCase, no specific word class: enum UnitType { Friend, Neutral }
Unity asset	Noun phrase in PascalCase

Don’t Get Same Value More Than Once

Instead of getting the same value more than once, store it in a local variable. Rationale: The C# compiler and Mono runtime are pretty stupid about optimizing this, so it will be inefficient. The code also gets easier to single-step and watch in a debugger. It avoids the problem with several long lines in a row. Putting the value in a variable may make its intent clearer.

```
// DON’T:
if(GetObject(someArgument) != null) {
    GetObject(someArgument).SomeMethod();
}
```

```
// DO:
SomeClass theThing = GetObject(someArgument);
if(theThing != null) {
    theThing.SomeMethod();
}

// DON'T:
foo.gameObject.transform.position = newPos;
foo.gameObject.transform.rotation = newRotation;

// DO:
Transform fooTransform = foo.gameObject.transform;
fooTransform.position = newPos;
fooTransform.rotation = newRotation;
```

Comments

First make sure the naming and code structure is as good as it can. Comment whatever is needed to make the code clear, after you have made sure that the code is as readable as you can without them.

Normally it's more valuable for comments that explain the reasoning behind the code, not how it works (as that should be obvious from the code).

Assume the person reading the code is at least an average programmer. Avoid "schoolbook comments" like:

```
// DON'T:
int x = 42;  // assign 42 to x
```

Do not keep unnecessary comments, e.g. Unity's default comment: `Update is called once per frame`.

Do not keep commented out code. Remove it instead. A single or couple of lines can be OK in some cases if it's useful for understanding the possibilities of the code. Log statements are sometimes useful to comment out instead of deleting (unless you have a proper logging system, which we don't).

Errors and Warnings

Treat warnings as errors. Do not keep code that generates compilation warnings. Fix it instead. For example, if you get a warning about an unused variable, remove that variable! Note that unfortunately third party code often gives warnings, which clutters the console. Do not fix these warnings in the third-party code. It's better to keep the original code. See below.

Third Party Code

When adding a Unity plugin, do not edit its code. Otherwise it's difficult to upgrade the plugin. If you need to change the code, create a copy, add a comment from where it's copied (e.g. `// Based on OVRBlaha from Oculus SDK v. 0.5.0`), and edit it.

When adding a plugin, make a note of its version number in the commit message, e.g. `Added Tango SDK version 0.2.1.`

When copying code from the web, add a comment about where you took it from, e.g. `Taken from http://www.stackoverflow.com/blabla`. This makes it possible to find the reasoning behind the code.

Fail Fast

If the game's state is not what the code expects, an error message should be shown as close as possible to the source of the problem. Fail fast, don't let invalid states linger.

Asserts

Use asserts to check that what the code expects is true. See The Unity Assertion Library. For example, if you *know* that a value has to be greater than zero, and it would be a programming error if it wasn't, you can write:

```
    Assert.IsTrue(value > 0, "Value should be above zero”);
```

Use AreEqual to check for equality. Note that the first argument is the expected value, the second is the actual value.

```
    Assert.AreEqual(5, frogCount, "Wrong amount of frogs");
```

Note that asserts are removed from release builds, so you can not depend on their side effects. Only use asserts to check for things that are programming errors. If an error can occur because of invalid user input, you should use normal error checks.

See the Assertions documentation for all available assertions.

Casts

Differentiate between the “as” operator and casting. I.e.

```
    x as Foo   // sets x to null if cast fails
```

vs.

```
    (Foo)x    // throws InvalidCastException if cast fails
```

Rule: Don't use the “as” operator to cast an object that *should* be of a specific type. E.g. do not do:

```
// Sets obj to null if Instantiate doesn't return what you
expect
GameObject obj = Object.Instantiate(...) as GameObject; //
DON'T!
```

The above will give a null pointer exception somewhere else in the code, which makes the error harder to track down. Instead fail fast with:

```
// Throws an exception (good!) if return value is unexpected
GameObject obj = (GameObject)Object.Instantiate(...);
```

Of course you should still use the "as" operator if it's expected that the object sometimes has another type.

Iterating

Do not use "for each" constructs when iterating over an array or List.

```
// DON'T!
foreach(var v in someArrayOrList) { ... }
```

Unfortunately, this is a performance problem as it allocates an iterator object. Instead do it the old-fashioned way:

```
for(int i = 0, len = someArray.Length; i < len; ++i) {
    var v = someArray[i];
    ...
}
```

Note that this example stores the array's length in the variable `len` to adhere to the Don't Get Same Value More Than Once rule.

Unfortunately, there seems to be no way to iterate over a Dictionary without allocating an iterator. If you iterate over a Dictionary often, you probably should consider another data structure.