Coding Conventions

Biased on Epic Games Coding Conventions, which can be found here:
https://docs.unrealengine.com/en-us/Programming/Development/CodingStandard

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If we decide to expose source code to mod community developers, we want it to be easily understood.
- Many of these conventions are actually required for cross-compiler compatibility.

The coding standards below are C++-centric, but the spirit of the standards are expected to be followed no matter which language is used. A section may provide equivalent rules or exceptions for specific languages where it's applicable.

Naming Conventions

For image, sound, and turn in files please prefix TGS_YOURNAME_ for example TGS_Timmy_FlyingMongose.png. Note that .c, .h, .lvl, .object, .spritesource, and .tileMap files are exempt from this rule as it will cause clutter.

- The first letter of each word in a name (such as type name or variable name) is capitalized, and there is usually no underscore between words. For example, `Health` and `PrimitiveComponent` are correct, but not `delta_coordinates`. If It is a private variable should start with a lower case letter. For example, `primitiveComponent`.
- Type and variable names are nouns.
- Method names are verbs that describe the method's effect, or describe the return value of a method that has no effect.

Variable, method, and class names should be clear, unambiguous, and descriptive. The greater the scope of the name, the greater the importance of a good, descriptive name. Avoid over-abbreviation.

All variables should be declared one at a time, so that a comment on the meaning of the variable can be provided. Also, the JavaDocs style requires it. You can use multi-line or single line comments before a variable, and the blank line is optional for grouping variables.

All functions that return a bool should ask a true/false question, such as `IsVisible()` or `ShouldClearBuffer()`.

A procedure (a function with no return value) should use a strong verb followed by an Object. An exception is, if the Object of the method is the Object it is in. Then the Object is understood from context. Names to avoid include those beginning with "Handle" and "Process" because the verbs are ambiguous.

Though not required, we encourage you to prefix function parameter names with "Out" if they are passed by reference, and the function is expected to write to that value. This makes it obvious that the value passed in this argument will be replaced by the function.

If an In or Out parameter is also a boolean, put the "b" before the In/Out prefix, such as `bOutResult`.

Functions that return a value should describe the return value. The name should make clear what value the function will return. This is particularly important for boolean functions. Consider the following two example methods:

```
// what does true mean?
bool CheckTea(FTea Tea);


// name makes it clear true means tea is fresh
bool IsTeaFresh(FTea Tea);
```

Class Organization

Classes should be organized with the reader in mind rather than the writer. Since most readers will be using the public interface of the class, that should be declared first, followed by the class's private implementation.

Comments

Comments are communication and communication is vital. The following sections detail some things to keep in mind about comments (from Kernighan & Pike *The Practice of Programming*).

Guidelines

- Write self-documenting code:

```
// Bad:
t = s + l - b;
// Good:
TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

- Write useful comments:

```
// Bad:
// increment Leaves
++Leaves;
// Good:
// we know there is another tea leaf
++Leaves;
```

- Do not comment bad code - rewrite it:

```
// Bad:
// total number of leaves is sum of
// small and large leaves less the
// number of leaves that are both
t = s + l - b;

// Good:
TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

- Do not contradict the code:

```
// Bad:
// never increment Leaves!
++Leaves;

// Good:
```

```
// we know there is another tea leaf
++Leaves;
```

Const Correctness

Const is documentation as much as it is a compiler directive, so all code should strive to be const-correct.

This includes:

- Passing function arguments by const pointer or reference if those arguments are not intended to be modified by the function,
- Flagging methods as const if they do not modify the object,
- and using const iteration over containers if the loop isn't intended to modify the container.

Example:

```
void SomeMutatingOperation(FThing& OutResult, const TArray<Int32>& InArray
)
{
    // InArray will not be modified here, but OutResult probably will be
}


void FThing::SomeNonMutatingOperation() const
{
    // This code will not modify the thing it is invoked on
}


Array<String> StringArray;
for (const String& : StringArray)
{
    // The body of this loop will not modify StringArray
}
```

Const should also be preferred on by-value function parameters and locals. This tells a reader that the variable will not be modified in the body of the function, which makes it easier to understand. If you do this, make sure that the declaration and the definition match, as this can affect the JavaDoc process.

Example:

```
void AddSomeThings(const int32 Count);
```

```
void AddSomeThings(const int32 Count)
{
    const int32 CountPlusOne = Count + 1;
    // Neither Count nor CountPlusOne can be changed during the body of th
e function
}
```

One exception to this is pass-by-value parameters, which will ultimately be moved into a container (see "Move semantics"), but this should be rare.

Example:

```
void Blah::SetMemberArray(Array<String> InNewArray)
{
    MemberArray = MoveTemp(InNewArray);
}
```

Put the const keyword on the end when making a pointer itself const (rather than what it points to). References can't be "reassigned" anyway, and so can't be made const in the same way.

Example:

```
// Const pointer to non-const object - pointer cannot be reassigned, but T
can still be modified
T* const Ptr = ...;


// Illegal
T& const Ref = ...;
```

Never use const on a return type, as this inhibits move semantics for complex types, and will give compile warnings for built-in types. This rule only applies to the return type itself, not the target type of a pointer or reference being returned.

Example:

```
// Bad - returning a const array
const Array<String> GetSomeArray();


// Fine - returning a reference to a const array
const Array<String>& GetSomeArray();


// Fine - returning a pointer to a const array
```

```
const Array<String>* GetSomeArray();

// Bad - returning a const pointer to a const array
const Array<String>* const GetSomeArray();
```

Code Formatting

Braces

Always include braces in single-statement blocks. For example.:

```
if (bThing)
{
    return;
}
```

If - Else

Each block of execution in an if-else statement should be in braces. This is to prevent editing mistakes - when braces are not used, someone could unwittingly add another line to an if block. The extra line wouldn't be controlled by the if expression, which would be bad. It's also bad when conditionally compiled items cause if/else statements to break. So always use braces.

```
if (bHaveUnrealLicense)
{
    InsertYourGameHere();
}
else
{
    CallMarkRein();
}
```

A multi-way if statement should be indented with each else if indented the same amount as the first if; this makes the structure clear to a reader:

```
if (TannicAcid < 10)
{
    UE_LOG(LogCategory, Log, TEXT("Low Acid"));
}
else if (TannicAcid < 100)
{
    UE_LOG(LogCategory, Log, TEXT("Medium Acid"));
}
else
{
    UE_LOG(LogCategory, Log, TEXT("High Acid"));
}
```

Tabs and Indenting

Here are the standards for indenting your code.

- Indent code by execution block.
- Use tabs, not spaces, for whitespace at the beginning of a line. Set your tab size to 4 characters. However, spaces are sometimes necessary and allowed for keeping code aligned regardless of the number of spaces in a tab. For example, when you are aligning code that follows non-tab characters.
- If you are writing code in C#, please also use tabs, and not spaces. The reason for this is that programmers often switch between C# and C++, and most prefer to use a consistent setting for tabs. Visual Studio defaults to using spaces for C# files, so you will need to remember to change this setting when working on Unreal Engine code.

Switch Statements

Except for empty cases (multiple cases having identical code), switch case statements should explicitly label that a case falls through to the next case. Either include a break, or include a falls-through comment in each case. Other code control-transfer commands (return, continue, and so on) are fine as well.

Always have a default case, and include a break just in case someone adds a new case after the default.

```
switch (condition)
{
    case 1:
        ...
        // falls through

    case 2:
        ...
        break;

    case 3:
        ...
        return;

    case 4:
    case 5:
        ...
        break;

    default:
        break;
}
```

Copyright Notice

Any source file (.h, .cpp, .xaml, etc.) for distribution must contain a copyright notice as the first line in the file. The format of the notice must exactly match that shown below:

```
// Copyright © 2019 DigiPen (USA) Corporation. All Rights Reserved.
```