Technical Document - Necromancer Game

Changelog

Version	Date dd/mm/yyyy	Description of Change
1.0	01/10/2019	Initial creation

Contents

Introduction

This document describes the technologies and tools used during the development of the **Necromancer VR Game,** along with the conventions for their usage.

Formatting

- Code examples are given in *italics*
- All code examples are written in C#
- ** indicates a comment used to explain examples

Technologies

- The core technology used for this project is the Unity game engine and Blender
    - Unity Version: 2019.2.3f1 , Windows 10 64 bit
    - Blender Version: 2.9, Windows 10 64 bit
- Custom functionality will be implemented in the C# (C-Sharp) programming language using Unity engine libraries
    - IDE: Microsoft Visual Studio Community 2017, Windows 10 64 bit
- Version control will be performed using Git
    - Remote Repo host: GitHub
    - Desktop Client: GitHub Desktop, Windows 10 64 bit

C# Conventions

Code Ordering

The following ordering of code should be applied in all class files:

Order:

1. Variables and auto-properties

2. Constructors
3. Methods
    a. private
    b. Protected
    c. public
4. Fourth: Properties (not including auto-properties)

Example:

```
public class MyClass
{
        ** Variables and auto-properties
        private int myNum = 0;
        public string MyWord { get; set} = “Hello”;

        ** Constructors
        public MyClass() { … }

        ** Methods
        private int MyPrivateMethod() { … }

        protected int MyProtectedMethod() { … }

        public int MyPublicMethod(){ … }

        ** Properties
        public int MyNum { .. }
}
```

Comments

Style

Prefix each class, method, variable, and auto-property with an XML Summary comment explaining their high level purpose or function. For methods, include descriptions of return type (if non-void) and parameters (if any). For comments on individual statements, use triple forward slashes (///).

There should be a single space between the triple forward-slashes and the first word of the comment text. Comments should end with a full stop.

All comments should be placed on the line immediately before the line or lines they describe, not at the end of it. They should also share the same level of indentation as what they describe, e.g. the comment for a method should be indented to the same level as that method's definition.

Strongly prefer verbosity and clarity over brevity. Use multiple lines if necessary.

For a list of relevant tags and XML commenting, see:

XML Documentation Comments

Example:

```
/// <summary>
/// Summary of myVar.
/// </summary>
private int myVar = 5;

/// <summary>
/// Summary of MyMethod.
/// </summary>
/// <param name="paramOne">Parameter one.</param>
/// <param name="paramTwo">Parameter two.</param>
/// <returns>Sum of paramOne and paramTwo.</returns>
private int MyMethod(int paramOne, int paramTwo)
{
        /// Comment on the statement below.
        Debug.Log("My statement.");

        /// Another single statement comment.
        return paramOne + paramTwo;
}
```

Braces

Opening and closing braces should be placed on their own lines

Example:

```
private void MyFunction()
{       ** Opening brace of method is on one line…
        ……
}       ** …and the closing brace on another
```

Blank Lines

No white space between opening brace and first statement

Example:

```
private void MyMethod()
{
        int _i = 0; ** No white space between method's opening line and the first
        statement
}
```

No white space between last statement and closing brace

Example:

```
private void MyMethod()
{
        int _i = 0; ** No white space between method's closing line and the last
        statement
}
```

Exactly one line of blank space between each statement

Example:

```
private void MyMethod()
{
        int _i = 0;
        ** Exactly one line of blank space between statements
        int _j = i;
}
```

Access Specifiers

Explicitly declare the access level of every class, method, property, and variable, regardless of whether the default is sufficient.

Example:

```
public class MyClass
{
        private int m_myVar = 5;

        public int MyVar { get; set; }

        private void MyMethod() { … }
}
```

Naming

Use Camelcase formatting for all variables names (including reference types, e.g. objects)

Example:

private int m_myVar = 5; ** the name ‘m_myVar’ is in Camelcase format, with the first letter of every word after the first capitalised (Camelcase is not applied to the m_ prefix)

private MyObject m_myObject ** same for reference types (again not applied to the m_ prefix)

Technical Document

Use Pascalcase formatting for all method, class, and interface names, e.g. MyMethod, MyClass, IMyInterface

Example:

> *public class MyClass* ** Class's name is in Pascal Case; the first letter of each word in the name is capitalised
> *{*
> *private void MyMethod() { … }* **Applies to method names as well
> *}*
>
> *Public interface IMyInterface* **

Apply the prefix 'm_' (a lowercase M followed by a single underscore) to the names of all global variables. Camelcase formatting begins after the prefix

Example:

> *public class MyClass*
> *{*
> *private int m_myVar = 5;* **Camelcase formatting is applied to all words in the name after the 'm_' prefix
> *}*

Apply the prefix "I" (a single uppercase I) to the names of all interfaces. CamelCase formatting begins

Apply the prefix "_" (a single underscore) to all local variables (including method parameters). Camelcase formatting begins after the prefix

Example:

> *private void MyMethod(int _myParam)* **Use the prefix for method parameters as well
> *{*
> *int _myVar = _myParam;* **CamelCase formatting is applied to words in the name after the '_' prefix
> *}*

All names must describe as much as possible the purpose of what they identify, e.g. for a variable that counts the number of steps taken, "m_stepsTakenCount" would be appropriate. Strongly prefer verbosity and clarity over brevity.

Example:

```
private int CalculateSum(int _a, int _b)
{
    return _a + _b;
}
```

Properties

Do not use public member variables. Use auto-properties instead. Any validation or operations to be performed on read/write should be added to the relevant accessors of these auto-properties.

Example:

public class MyClass
{
public Int MyVar { get; private set; } **This declares an int variable, identified by 'MyVar', and defines that it can read publically, but only set privately
}

Unity Conventions

Project Structure

The structure of the Unity project directory is described below used during implementation is described below (** represent descriptive comments; not part of directory name):

- ➔ Assets (root directory)
  - ◆ Animations **Contains animations and related files
  - ◆ Audio Assets **Contains audio assets, separated into subdirectories by type

- Music **Contains music files
- Sound Effects **Contains sound effects files
- Voice **Contains voice audio files

◆ Old Assets **Contains assets that are no longer used in the project
- Old Visual Assets **Contains visual assets that are no longer used in the project

◆ Prefabs **Contains Unity prefabs
- Particle Effects **Contains particle effect prefabs

◆ Scenes **Contains Unity scenes

◆ Scripts **Contains C# scripts
- Editor **Contains scripts that use the Unity.Editor namespace (i.e. that include *using Unity.Editor*.) These scripts will cause builds to fail if they are not placed in this directory
- Events **Contains event systems scripts
- Interfaces **Contains interface scripts
- Managers **Contains '-Manager' scripts
- Models **Contains '-Model' scripts

◆ Testing Scenes **Contains scenes used for testing purposes

◆ Testing Scripts **Contains scripts used for testing purposes

◆ Visual Assets **Contains visual assets, seperated into subdirectories by type
- PSDs **Contains Adobe Photoshop PSD (.psd) files
- Sprites **Contains Unity sprite assets.
- Blender Files **Contains Blender scenes/models.
- Models **Contains models that have been exported from Blender into an FBX format.
- Materials **Contains the materials used to texture objects within Unity.

Alternatives and Add-Ons

Text Mesh Pro

Where possible, use Text Mesh Pro alternatives to standard Unity versions of objects, e.g. Text Mesh Pro - Text objects instead of Text objects.

Naming

Script Files

Use Pascalcase formatting for all script files

Example:

MyScript **The first letter of every word in the name is a capitalised

GameObjects

For all non-GUI GameObject names, begin each word in the name with a capital letter and separate each non-hyphenated word with a space.

Example:

My Game Object

My Power-Up

For UI GameObject names, apply a prefix indicating the type of the UI element, followed by an underscore, then the name formatted in Camel Case. The following are valid prefixes:

lbl_ **Text or Text Mesh Pro - Text

Example:

lbl_myLabel

txt_ **Input Field or Text Mesh Pro - Input Field

Example:

txt_myText

sld_ **Slider

Example:

sld_mySlider

img_ **Image

Example:

img_myImage

btn_ **Button

Example:

btn_myButton

tgl_ **Toggle

Example:

tgl_myToggle

scl_ **Scrollbar

Example:

scl_myScrollbar

drp_ **Dropdown

Example:

drp_myDropdown

pnl_ **Panel

Example:

pnl_myPanel

File Types

Image Files

All imported image files should be in PNG (.png) format where possible

Audio Files

All imported audio files should be in WAV (.wav) format where possible

Code

At most one outer class per script file. Any number of inner classes are permitted.

Example:

```
public class OuterClass
{
	private class InnerClass
	{
		……..
	}

	private class AnotherInnerClass
	{
		……..
	}
}
```

Variables that are intended to be editable in the Unity Inspector must be declared as *private* with the *SerializeField* Attribute. Do not use public variables for this purpose.

Example:

```
	[SerialiseField]
	private int m_myVar = 0; **m_myVar will now be visible in the Inspector
```