Unity: стандарт кодирования

Иванова М., Бурков Э.

Оглавление

Общие правила

1. Лучше писать код медленнее, то более качественно.
2. Лучше писать код медленнее, но сделать его более легким в поддержке
3. Старайтесь сократить объем кода там, где это возможно. Но важно, чтобы сокращения не приводили к усложнениям для понимания другими разработчиками.
4. При компиляции готового кода не должно быть warning'ов.

Оформление кода

1. Фигурные скобки

1.1. Фигурные скобки обязательно должны располагаться каждая *на отдельной строке*. Открывающая скобка всегда находится в начале строки, следующей за той, которая открывает блок. Закрывающая должна быть в конце блока визуально под открывающей.

```
if (someExpression)
{
    DoSomething();
}
else
{
    DoSomethingElse();
}
```

1.2. Все содержимое блока между фигурными скобками должно быть выделено *4 пробелами* (табы нельзя использовать)

1.3. При использовании оператора *switch* нужно использовать фигурные скобки. Выражения в *case*, содержащие несколько строк, *должны быть также обрамлены фигурными скобками*.

```
switch (someExpression)
{

   case 0:
      DoSomething();
      break;

   case 1:
      DoSomethingElse();
      break;

   case 2:
      {
         int n = 1;
         DoAnotherThing(n);
      }
      break;
}
```

1.4. Фигурные скобки *должны быть* даже если *блок содержит одну строку кода*. Это позволяет сделать код более читабельным.

1.5. *Тело «пустых» методов* должно занимать *две строки*

```
void EmptyMethod()
{
}
```

1.6. Возможно написание свойства (get/set конструкции) в 1 строку, если оно используется как автоматическое.

```
public int Var { get; set; }
```

2. Пробелы

2.1. Одинарные пробелы должны использоваться *после запятой между аргументами функции или метода*

```
Console.In.Read(myChar, 0, 1);
```

2.2. Одинарный пробел используется *перед сторожевым условием на исполнение блока кода*

```
while (x == y)
```

2.3. Одинарный пробел *перед и после оператора сравнения*

```
if (x > y)
```

2.4. Одинарные пробелы используются для *разделения арифметических операций и операндов.*

```
int value = (number + 2) * 10;
```

2.5. Нельзя использовать пробелы *после скобок или между аргументами функции и запятыми после них*

```
CreateFoo(myChar, 0, 1)
```

2.6. *Нельзя* использовать пробелы *между именем функции и скобками*

```
CreateFoo()
```

2.7. *Нельзя* использовать пробелы *внутри квадратных скобок*

```
x = dataArray[index];
```

2.8. *Нельзя* использовать пробел *между производными типами*

```
var list = new List<int> ();
```

3. Именование объектов

3.1. Имена объектов должны быть *легко читаемы* и *соответствовать назначению объекта*. Нельзя использовать числа как имена параметров. Если непонятно, как назвать параметр *унарной функции*, то можно использовать имя *value*. Для параметров *бинарной функции* можно использовать *left и right*.

3.2. *Нельзя* использовать *венгерскую нотацию* при именовании объектов.

3.3. *Нельзя* использовать *префиксы перед именами переменных*, *перечислений и классов* (_, m_, s_ и т.д.). Если требуется *отделить локальные переменные от параметров методов*, *используйте this*. Используйте this только в случае крайней необходимости.

3.4. Для имен *локальных переменных и параметров методов* используйте *camelCase*.

3.5. Для *имен методов, свойств, событий, типов, пространств имен, перечислений и имен классов* используйте *PascalCasing*.

3.6. Используйте *префикс "I"* перед *именами интерфейсов*.

3.7. Имена *классов и структур* должны быть *существительными*. В именах *интерфейсов* могут использоваться *прилагательные и существительные*. В именах *методов* используются только *глаголы и глагольные выражения.*

3.8. *Нельзя* называть методы, начиная *со слова get*, если это не стандартные геттеры.

3.9. Имена *булевских переменных* должны быть только *в утвердительной форме*. Можно использовать "Is", "Can" или "Has" в начале имени там, где это имеет смысл.

3.10. Имена ресурсов должны заканчиваться на то существительное, которое говорит о цели использования данного ресурса. (MyRes1Texture, MyRes2Model, …)

3.11. Глобальные переменные именуются с использованием *camelCase* и начинаются со слова *global*. (globalMyVar, globalValue)

3.12. Имена констант должны быть в UpperCase формате. Если наименование многосложное используется '_' для разделения членов. (CONSTANT, MY_CONST_VAR)

3.13. В наследовании/имплементации порядок наследуемых/имплементирующих классов должен быть алфавитным. (class A : IAInterface, IBInterface, ICInterface)

4. Форматирование параметров функции

4.1. Если параметров функции слишком много и их требуется перенести на другую строку, то новая строка параметров должна начинаться там же, где начинаются параметры на строку выше.

```
CreateLine(format, foo,
           bar, baz)
```

4.2. Если параметры функции слишком много, то их все можно перенести на новую строку. При этом открывающаяся скобка должна остаться на той же строке, где имя метода.

```
CreateLine(
   format, foo, bar, baz)
```

4.3. При переносе параметров на новую строку запятая должна оставаться в конце строки, а не в начале.

```
CreateLine(format,
           foo,
           bar,
           baz)
```

5. Организация файлов

5.1. Исходные файлы могут содержать только один публичный класс. В противном случае файл требуется разделить на несколько (по количеству публичных классов). В публичном классе могут быть несколько вложенных классов.

5.2. Имя исходного файла должно совпадать с именем публичного класса, определенного в этом файле.

5.3. Все члены класса должны быть отсортированы по алфавиту и разбиты на смысловые группы (поля, конструкторы, свойства, события, методы и т.д.)

```
using System;
using UnityEngine;

public class MyClass : MonoBehavior
{
    // fields
    int foo;

    // properties
    public int Foo { get { … } set { … } }

    // methods
    void MyMethod(int number)
    {
        int value = number + 2;
        Debug.Log(value);
    }
}
```

6. Длина строки

6.1. Максимальная длина строки в C# равна 180 символам.

7. Инициализация новых объектов

7.1. При создании новых объектов класса значение полей задаются либо с помощью уже созданного конструктора, либо отдельно определяя значение каждого поля. Нельзя определять новое тело конструктора при создании нового объекта.

8. Комментарии

8.1. Комментарии должны описывать основную идею следующих за ним строк, суть алгоритма или логический поток. Писать комментарии надо так, чтобы любой сторонний пользователь должен, прочитав комментарии, понять работу основных функций и операций.

8.2. Для *однострочных* комментариев используется *//*. По возможности комментарии должны *предшествовать коду*, к которому они относятся.

8.3. Если комментарий *достаточно короткий* (содержит 2-3 слова), то он может быть записан *после выражения*, к которому он относится*, в той же строке*.

8.4. Комментарии, *занимающие несколько строк*, должны быть оформлены следующим образом:

```
/*
 * Blah
 * Blah again
 * and another Blah
*/

//
// Blah
// Blah again
// and another Blah
//
```

8.5. При описании функции можно использовать: *емкий комментарий*, описывающий цель функции, или *подробный комментарий*, описывающий параметры функции и возвращаемое значение.

```
/* Func name.
 * > 'Var var1' – var for fun.
 * > 'int num' – num of cool.
 * < nothing/'List' - cool list of things
 */
```

Использование GitHub

1. *Перед вливанием* нового локального кода в удаленный репозиторий следует сначала сделать *pull текущего состояния рабочей ветки* и локально *разрешить все возникнувшие конфликты*. Это позволяет избежать случайного удаления чужого кода. Если разработчик не уверен, какие изменения нужно оставить, надо посоветоваться с коллегами.
2. При *commit'e* обязательно надо указывать *однострочное описание сделанных изменений* (summary), а затем *подробно описать, какие работы были сделаны*. Это позволяет легче отслеживать на github'е сделанные изменения.
3. Новые *feature-ветки* создаются только *после одобрения Натальи Дмитриевны* для больших по объему задач, которые никак не пересекаются с другими задачами, находящимися сейчас в работе. После окончания работы в ветке разработчик делает *pull*

request с feature-ветки в master-ветку, чтобы остальные участники могли видеть сделанные изменения.

4. *Не выкладывайте* в удаленный репозиторий *нерабочий код, который мешает работе других участников проекта.*

Использование Unity

1. Лучше пере-использовать выделенные ресурсы, вместо того, чтобы создавать новые.

Источники

http://wiki.unity3d.com/index.php/Csharp_Coding_Guidelines

https://prezi.com/povbmlr5-w8j/coding-standards-for-game-development-in-unity3d/

http://www.mono-project.com/community/contributing/coding-guidelines/

https://msdn.microsoft.com/en-us/library/ms229002.aspx