C# Guide for simulation framework

This is a quick and rather informal guide on how C# differs from other programming languages and how to use it. It only covers the language features that are used in Continuum, our simulation framework. If anything remains unclear, feel free to send an email.

NOTE: any ***bold italic*** words are C#/programming terms and thus can be searched for more information regarding the topic.

For C# IDEs/code environments we recommend Rider or Visual Studio, but Visual Studio Code also works with a C# dev kit extension. For running C# code, you need to have an entry point for your project, in the code we provided the projects: 'BasicImplementation' and 'GrowthGridImplementation' and '3DRenderer' has such an entry point and thus can be run. The simulation itself and the Implementations projects are libraries and cannot be run on their own. C# has a JIT (Just-In-Time) compiler, meaning you can run the code before doing any compiling and then the compiling happens when the program is running, which also means that pressing the run button on your IDE will always be the latest version. C# can also be compiled ahead of time by ***building*** it to a .exe which gives more performance. C# can also be run in Debug mode by all IDEs, doing so will allow you to set ***breakpoints*** in the code. When a breakpoint is reached, the IDE will stop running your code and give you an overview of the current program state. Note that the line the breakpoint is set at is not yet run, that will happen after doing a ***step***. Using an IDE you can ***step over*** a line to execute it and go to the next line, or ***step into*** it to go into the execution of that line. You can also ***continue*** the program to leave the step-by-step code execution.

Now let's get into C# itself. C# Object Oriented Programming language built upon C++ and shares some of the semantics of it. However, it is most similar to Java in how it works and reads. C# has ***projects*** which contain files with code. When you run a C# program, you run a project. A ***solution*** is basically a group of projects, you cannot run an entire solution, only the projects within it. All the code for the framework is within the same solution to group the code together. C# does not have pointers from the user end. Everything that is a class in C# is ***passed by reference*** (meaning it is a pointer to the data) and all structs are ***passed by value*** (meaning the variable stores the data itself. This distinction is important for how values are passed to- and returned from ***methods***.

A ***method*** is the C# term for a function.

A ***property*** is the C# term for a 'normal' variable (i.e. int, float, string, ect).

Methods in C# have a return type, ***void*** means nothing is returned and otherwise it is the stated type that is returned.

Methods and properties have an ***access modifier*** which describes who and what is allowed to access it. ***Public*** means anything can access it, ***private*** means nobody except itself can access it. ***Protected*** means only itself and ***subclasses*** of it can access it (more on ***inheritance*** later) and ***internal*** means it is public within the project, but it cannot be accessed from different projects.

As C# is OOP it has inheritance, meaning one class can derive from another. In this example TestOrganism has inherited from Organism, where TestOrganism is the ***subclass*** and Organism is the ***superclass***.

```
public class TestOrganism : Organism
```

The advantage of inheritance is that TestOrganism can now use all of Organism's methods and properties that are not private.

Sometimes a method or property in TestOrganism might say ***override***. This means that it is changing the original behaviour of a method. If a method in Organism contain the ***keyword*** (any word in from of the method name) ***virtual*** then it can replace the function with its new version using override. The original version of that function can still be called in the new method by calling base.[METHODNAME](). Override can also be used on an ***abstract*** method. Abstract methods do not contain any contents but are required to be filled in by a subclass (which is what we use a lot, as it guarantees that a certain function exists when we need it). This is an example of an abstract method:

```
public abstract void Step();
```

Something that looks a lot like an abstract class is an interface, interfaces do not contain any code, but only contain requirements for a class that would inherit it. The benefit of this is that the programmer can request an interface in a section of the code and be certain that every listed method exists. Example:

```
public interface IMinimumBoundable
{
    public Mbb GetMbb();
    public void SetMbb(Mbb newMbb);
}
```

Some quick extra note about C#:

In C# it is typical to have 1 class per file, but it is possible to have many classes within a single file. In C# ***using*** is the word for import. Typical naming rules are PascalCase for method names and public properties, and camelCase and _camelCase for all other properties (however we mostly due not use _camelCase due to personal preferences). C# is also very lenient with referencing other files and classes, unlike javascript it is

possible to have 2 files reference each other and to have circular references. This only breaks on importing libraries/other projects, but within a project everything is possible.

The keyword ***static*** can be added to classes/methods/properties, doing so means that the value is consistent for all version of that class. This also means that you do not have to create an instance of the class to call its methods/properties, like so:

```
double pi = Math.PI;
```

The keyword ***const*** can be used for properties with it enforcing that the value never changes. A const value also works like a static value, so the Math.PI value above is actual a const in this case, but you get the idea.

The keyword ***async*** allows a function to be run simultaneously by multiple ***Tasks*** (which get automatically distributed over available ***threads***. However, a function being async does not in itself mean that it is being run multithreaded, just that it can be it requested by the programmer.

The normal way to set a property is through:

```
private Vector3 position;
```

However you might sometimes see:

```
public float Size { get; private set; }
```

These are called a ***getters*** and a ***setters*** respectively. In this situation I can get the value Size from within any other class, but I can never change the value, as that is only allowed by the Organism itself (because the setter is private). If a setter is completely missing, then it can only be set in the ***constructor*** of the class. The ***access modifier*** of the getter is that of the entire property (so public in this case).

A special type of variable is ***var***, this cannot be a property as it can only be made within a method not set for an entire class. Var works like a wildcard, you can pass it any type of value (except void) and it will become it, however, the value type cannot change after that (so you can't first make it an int and then turn it into a float). This is useful when you get a very long type name that is returned, or if you do not yet know what is returned, after which you can turn it into the correct type.

You might see ***IEnumerable<Organism>*** a lot in our code, this is an interface for anything that can be looped over. So it could be an ***array***, or ***List***, or ***LinkedList***, or something else. Looping over something can be done with a ***for*** loop or a ***foreach*** loop. The ***<Organism>*** here means that the enumerable type contains only Organisms as in C# lists containing multiple types is not allowed (although it can contain any subclass of the type, so you can loop over ***object*** to allow almost anything into an enumerable type).

Sometimes you might see a type with ***?*** after it, for example:

```
public abstract Organism? NearestNeighbour(Organism organism);
```

The ***?*** means that that type can be null and will throw an error if you try to read it, so you first need to check organism != null before accessing its values.

With it you might sometimes see a ***!***, the ***!*** does not do anything in itself, but it lets the IDE know that you know that something nullable is definitely not null, so it will not show a warning (but it will still crash if it turns out to be null when you try to access its values).

There is a different type of ***?*** in C# that you should not get the previous one confused with and that is the ***ternary conditional operator*** as it is called. It is used as a shorthand for an if-else statement.

```
int a = 5;
int b = 10;
int c = looping ? a : b;
```

Here value c is 5 if looping is true and 10 if it is false. The ***:*** is what separates the true-value from the false-value.

Something our code also contains is casting.

```
float lambda = (float)(2 * Math.PI * Random.NextDouble());
```

Here the ***(float)*** is used to indicate that the the result of what comes after (which is of type double in this case) should be turned into type float. This is not always possible as C# needs to know that a standard logical conversion is possible and it will still throw an error if something incorrect happens (for example casting a float with a fraction to an int will return an error as an int cannot contain fractions).

Finally, here we will list a few useful classes for when you want to program:

List, Random, Math, MathF, File, Directory, Path, Vector3, Vector2 (note that this and vector3 have multiple version from different libraries and you will have to use the same one as the simulation library uses), Dictionary (C# version of HashMap).

A quick final note to avoid confusion while you are programming. File interactions in C# typically use relative paths (but can also use absolute paths) starting from the ***bin*** folder. This folder can be found within the folder where you stored you C# solution.