SD325674

Leveling Up Your 3D C# Programming Skills Using Functional Programming

Christopher Diggins
Head of Research VIM AEC

Learning Objectives

- use functional programming techniques in C# to write less code
- use immutable programming techniques to make code more robust and easier to parallelize
- more quickly break down hard 3D programming problems into simpler ones
- write more reusable code

Description

Many Autodesk products support a C# API, Revit software, AutoCAD software, 3ds Max software, and Maya software, just to name a few. One of the advantages of C# is excellent support for pure functional programming. In this class, I will teach you how we can use functional programming techniques to transform what has traditionally been considered complex problems for high-performance, 3D graphics programming into relatively simple problems that are easier to solve and require less code. We will demonstrate this using a number of examples from open-source libraries that are being used to solve some real-world, large-scale geometry processing problems. We will also share various insights from a career of more than 25 years of professional software development.

Speaker(s)

My name is Christopher Diggins. I have been programming personal computers for fun since 1984, and professionally since 1995. I am the Head of Research at VIM AEC where my focus is on the efficient processing of large data sets and geometric computation. I am very passionate about the developer experience, and always thinking what can be done to help make myself and other programmers more productive and to advance the state of the art of software development.

Writing Better Software More Efficiently

"The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer."
– Edsger W Dijkstra (1972 Turing Award Lecture)

Programming personal computers has come a long way than when I first started in 1984, but it still seems to be a lot harder for most than it should be. Collectively the software engineering industry seems to acknowledge have observed that different software solutions to the same problem can indeed vary greatly in terms of elegance and complexity.

As I have gained more experience in different programming paradigms and techniques, I learned that sometimes finding the right way to look at a problem makes all the difference, and that applying the right abstractions can make a significant reduction in the amount of code we have to write and maintain, and the ease at which we can make changes to our code.

My experience has been that by rigorously adopting a more declarative approach to programming, specifically by applying lessons from the functional programming paradigm, leads to code that is naturally more compact, expressive, robust, and easier to maintain.

What is Functional Programming?

Declarative Programming

Functional programming is a subset of declarative programming and as such inherits the benefits. Declarative programming is paradigm where the desired end-result of the software or computation is described formally in terms of the problem domain rather than as a series of steps to achieve an end-result.

Example of declarative programming:

- Constraint programming
- Theorem provers
- Combinator parsing libraries
- Regular expressions
- Query languages (SQL, XPath, LINQ)
- Dataflow programming
- Hardware description
- GUI systems (XAML)
- Template systems

The Code is the Specification

The big advantage of a declarative programming approach is that the code can become the specification, side-stepping the classic problem of always having to keep up to date a specification document and code. In some cases, the declarative components the map directly to the problem domain can be maintained by subject matter experts with no programming experience or updated via tools.

Functional Programming

Functional programming is a style of programming that emphasizes the usage of pure mathematical functions overs procedures and subroutines. Consequently, this paradigm emphasizes immutable data structures, functions as first class values, and the lack of side effects. The result is that software components are easier to refactor and reason about.

Pure Mathematical Functions and Referential Transparency

A pure mathematical function (or pure function) has the property of being referentially transparent (https://en.wikipedia.org/wiki/Referential_transparency), that is to say:

- It has no side effects
- Given the same inputs it will always evaluate to the same value

Subroutine versus Function

Traditionally subroutines and procedures consist of a series of statements that describe how to perform a computation on some share state whereas a function describes how to transform some input data into a new form, without depending on shared state, as a sequence of expressions.

Pure Functions may Use Imperative Programming Techniques

Within its own scope, pure functions may indeed use imperative programming constructs like variable rebinding, loops, and even jumps, but this is done simply for the ease of expressing and communicating intent: the observable result of a pure function must still remain referentially transparent.

Using Mutable Data Structures to Construct Immutable Data Structure

Sometimes it can be more convenient to create a mutable data structure (e.g. System.StringBuilder) that is designed to facilitate the efficient construction of an immutable data structure (e.g. System.String). This is a useful way to isolate the effects of imperative code, and retain the benefits of immutable data structures in the bulk of the program logic.

Functional Programming Emphasizes Upfront Design of Data Structures

---

In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful […] Torvalds, Linus (2006-06-27)

Since pure functions can only create new values from existing values, writing software in a functional style requires one to thing more about the form of the data flows between the

functions. If you are familiar with Git, the implementation is very similar to a pure functional data structure [https://blog.jayway.com/2013/03/03/git-is-a-purely-functional-data-structure/].

Functions as Values

In functional programming, like mathematics, a function is also a value like a number. It can be passed to a function as an argument or returned from another function. You can construct new one dynamically. This last point is very important since it is what distinguishes functions in a functional programming sense from function pointers in C/C++.

Fundamental Operators on Functions

Given that functions are operators there are some basic operations we can perform on them. Some examples are:

- function abstraction - the definition of a new function
- function application – calling or applying a function to its inputs
- partial application – creating a function
- function composition – given two functions (f and g), returns a function (h) that given an input (x) is equivalent to the application of f to the application of g to x, in other words h = f . g h(x) = f(g(x)
- function derivative - the sensitivity to change of the function value (output value) with respect to a change in its argument

Lambda Calculus

It is interesting to note that in the Lambda Calculus an entire computational system, including arithmetic, can be created from functions alone and the fundamental operations of function abstraction and application.

Functional Programming Examples

Several locations where you already may be using functional programming:

- Task based parallelism
- LINQ
- Shaders
- Event handlers
- Foreach
- Github

Writing less code and reusing more code

Code is the Enemy

As a software developer matures and gains more experience, they start to learn that the goal of programming is to use as little code as possible.

Programs with more code in general:

- have more bugs
- are harder to understand, review, and debug
- take longer to refactor
- take longer to extend

Steve McConnell on diseconomies of scale in software development:

> **Project size is easily the most significant determinant of effort, cost and schedule [for a software project].***
>
> People naturally assume that a system that is 10 times as large as another system will require something like 10 times as much effort to build. But the effort for a 1,000,000 LOC system is *more* than 10 times as large as the effort for a 100,000 LOC system.
>
> [Using software industry productivity averages], the 10,000 LOC system would require 13.5 staff months. If effort increased linearly, a 100,000 LOC system would require 135 staff months. But it actually requires 170 staff months.

Here's the single most important decision you can make on your software project if you want it to be successful: *keep it small.* Small may not accomplish much, but the odds of outright failure-- a disturbingly common outcome for most software projects-- is low.

Tips to make your code more reusable

Code is maximally reusable when it:

- follows the robustness principle - conservative in what is returned, and liberal in what is accepted (https://en.wikipedia.org/wiki/Robustness_principle)
- avoids preconditions - any predicate that must always be true just prior to the execution of some section of code
- avoids side effects – including modification of the arguments
- respects referential transparency
- only returns one value
- has small functions
- follows the single responsibility principle https://en.wikipedia.org/wiki/Single_responsibility_principle
- avoids unnecessary coupling
- follows the open/closed principle
- following the dependency inversion principle
- are expressed in the problem domain rather than the solution domain

Robustness Principle (Postel's Law)

Postel's law can be stated as "Be conservative in what you send, be liberal in what you accept.". So in concrete terms these means to not require a `List<T>` as an argument when an `IEnumerable<T>` will do. However, do not accept an `IEnumerable<T>` when you need to have constant time random element access, in which case you would use an `IArray<T>`.

Respecting the Open-Closed Principle

The open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code. Functions that take functions as arguments naturally follow the open-closed principles.

Higher-Order Functions

A higher-order function is a function that either accepts a function as an argument, or returns a function. A function that does a general task and does that in terms of a function argument to describe how to do a sub-task, can be used in a wider set of cases.

Consider the interface of an immutable array called `IArray`.

```
/// <summary>
/// Represents an immutable array with expected O(1) complexity when
/// retrieving the number of items and random element access.
/// </summary>
public interface IArray<T>
{
    T this[int n] { get; }
    int Count { get; }
}
```

The following higher order function can be used to implement all conceivable instances of an `IArray`:

```
/// <summary>
/// Creates an IArray with the given number of items,
/// and uses the function to return items.
/// </summary>
public static IArray<T> Select<T>(this int count, Func<int, T> f)
    => new FunctionalArray<T>(count, f);
```

Some examples of the types of algorithms that you can build once you have a higher-order function:

```
public static class FunctionalArrayDemo
{
    public static IArray<T> Repeat<T>(T x, int count)
        => count.Select(i => x);

    public static IArray<int> Range(int count)
        => count.Select(i => i);

    public static IArray<T> Reverse<T>(this IArray<T> xs)
        => xs.Count.Select(i => xs[xs.Count - 1 - i]);

    public static IArray<U> Select<T, U>(this IArray<T> xs, Func<T, U> f)
        => xs.Count.Select(i => f(xs[i]));

    public static IArray<T> SubArray<T>(this IArray<T> xs, int from, int count)
        => count.Select(i => xs[i + from]);

    public static IArray<T> Stride<T>(this IArray<T> xs, int from, int stride)
        => (xs.Count / stride).Select(i => xs[i * stride + from]);

    public static IArray<T> Take<T>(this IArray<T> xs, int count)
        => xs.SubArray(0, count);

    public static IArray<T> Skip<T>(this IArray<T> xs, int count)
        => xs.SubArray(count, xs.Count - count);

    public static IArray<V> Zip<T, U, V>(this IArray<T> xs, IArray<U> ys, Func<T, U, V> f)
        => xs.Count.Select(i => f(xs[i], ys[i]));
}
```

Equivalency to Interfaces

An astute object-oriented programmer will make the observation that most of what you can do with higher-order functions can be done via interface albeit with more boilerplate code.
The question then should be raised, when do we use interfaces and when do we use functions?

- Interfaces should be reserved to describe and classify data types
- functions to describe computations
- classes to implement interfaces or to pass structured data

Working in the Problem Domain not the Solution Domain

> *"[A good architect] focuses as much on the problem to be solved and the various forces on the problem as he does on the solution to the problem. [The IT industry has] a tendency to focus on the solution." (Gamma et al. (Gang of Four), Design Patterns.)*

Code in which the abstractions are expressed in terms of the problem domain instead of the solution domain are easier to understand, debug, and analyze.

A natural result of coding in a functional style is that the constraints, inputs, and outputs of the problem domain can be more easily expressed.

Functions that take Functions as Arguments instead of Interfaces

Consider a function that takes an interface as an argument with only one function (e.g. `IEquality`). E.g. “CountInstancesOf()”.

Making code more robust and easier to parallelize

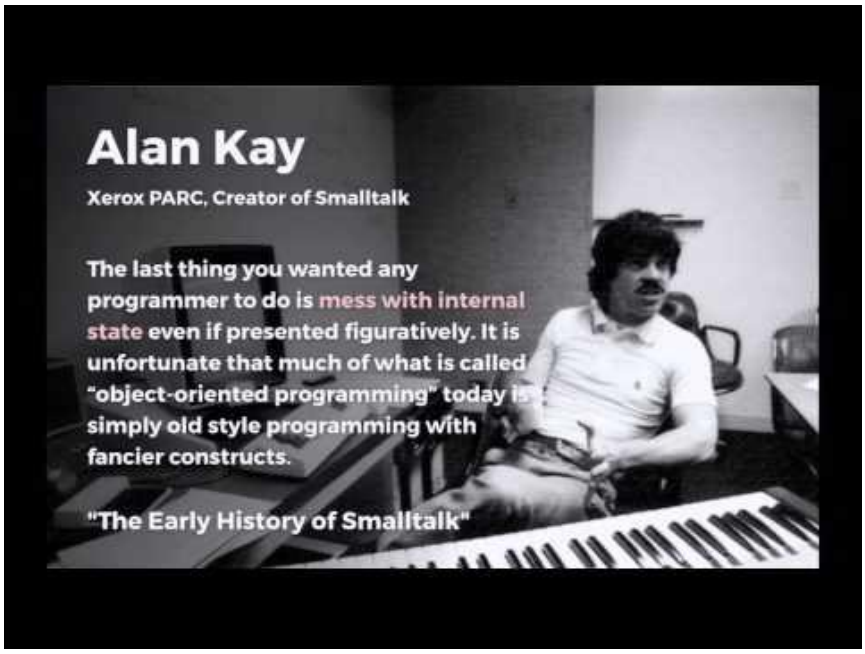
Simplicity is prerequisite for reliability. - Edsger Dijkstra, 1975 from How do we tell truths that might hurt?

It all comes down to Immutability

Code is hard enough to reason about let alone when your objects can change over time. When objects can’t change, you don’t have to test the order in which things happen. I can’t overstate how much of a difference it makes when objects and interface are immutable.

Object Oriented Programming Encourages Code Coupling

This is a not something you will hear often, but traditional method of teaching object-oriented programming, is flawed. It encourages code coupling. All methods implicitly share the state of the object. From a historical perspective this is far better than using shared global state, but today we should question whether mutable objects (object which change state) are really a good idea.



ALAN KAY RECOMMENDING NOT MESSING WITH INTERNAL STATE

Easier Parallelization

Concurrency in the presence of shared state is a complex problem. Most books about concurrency are techniques for dealing with concurrent read/write access to mutable data structures, and shared data. Eliminate shared state and the problem becomes simply how to design efficient immutable data structures.

Breaking down hard programming problems into simpler ones

Simplicity is a great virtue, but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

- *Dijkstra (1984) On the nature of Computing Science (EWD896).*

Abstractions

Edsger Dijkstra said that "the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer". I would go so far as to say that the fundamental act of programming is the craft of creating, naming, and combining abstractions.

Some examples:

- variables – instead of using registers and the stack
- functions and subroutines – better than “goto” and “gosub” from BASIC
- subroutine arguments – better than
- naming data types (structs, classes, interfaces)
- modules and namespaces
- projects and libraries
- executables
- services

Understand the Problem before Attempting a Solution

Too often programmers launch into a computer program, which is a solution, before properly understanding and modeling the problem domain. If more care is taken to understand that problem domain and to attempt to map it to code, the solution often falls from it naturally.

Say what you want, not how to do it

The imperative style of programming encourages an approach of “how to perform a computation” as a series of statements that modify some internal state. Whereas the functional style emphasizes the computation as a series of functional transforms from some input to another output.

An Iterative Approach to Building a Functional Program

First ask yourself this question: what would a function look like to solve my problem? Try to write a function signature that would solve that problem, possibly with imaginary types for the inputs and outputs. Then write what the body of that function, by breaking it into subproblems. Each step as a computation that transform the input into an output, using imaginary functions and types as needed. Repeat this process for each function and invent constructors for each data type. Eventually you fill out all of the functions, and everything just works.

Easier to Reason About

Pure mathematical functions are easier to reason about formally using analytical methods, but also informally by people just trying to read and understand code. When you look at a pure function, you know what it does based on what it calls, and what it does with the return values. You know it isn’t quietly changing the values of the arguments you pass it. That is worth gold for someone maintaining code.

Compose Abstractions, Don’t Rewrite Them

One way to think about the open-closed principle is that we should not be rewriting abstractions (e.g. functions, classes) to match our specific problem at hand, instead we should try to compose existing ones together. If this is not possible with our existing abstractions.

Simplicity is Hard but the Payoff is Huge

"Simplicity is hard work. But, there's a huge payoff. The person who has a genuinely simpler system - a system made out of genuinely simple parts, is

going to be able to affect the greatest change with the least work." – Rich Hickey

---

This is particularly relevant when trying to solve problems using higher order functions. I'll be the first to admit that it isn't easy to find the correct abstraction using immutable structures, or to get your head around complex LINQ queries, but it is worth it! Ultimately you need to be willing to trade off some of the complexity at the level of implementation, to gain the benefits of overall system simplicity.

Avoid Introducing Unnecessary Complexity

As a software developer, you are your own worst enemy. The sooner you realize that, the better off you'll be. - Jeff Atwood

---

The source of most complexity is from programmers, not clients. This comes becomes programmers are trained to be acutely aware of the technical problem domain of computer programming: memory management, performance, platform dependence.

My advice to programmers is this: think like client, not programmers, when programming. Always look for the "simplest thing that could possibly work". Validate early and often with the customer of the software. Let them come back with additional requirements. Too often, additional complexity is added to the system because of artificial requirements imagined by a developer, which are often added at the expense of the client's desired feature set.

This advice could be summarized by the advice of avoid creating a YAGNI ("You Aren't Going to Need It"), or simply by following the KISS principle (Keep It Super Simple).