Level up your C# Skills
With Functional Programming

Christopher Diggins
Head of Research at VIM AEC
Christopher Diggins
Head of Research

My area of focus is on efficient data formats for the transmission and rendering of design and manufacturing data, as well as applied computational geometry based on functional programming.

In previous lives I have been a principle engineer at Autodesk, a research assistant in the McGill Neuroscience dept., and a musician in the Canadian Ceremonial Guard.
A personal history of learning programming

- Gosub – Basic
- Procedures – Logo
- Functions – Pascal
- Modules – Pascal
- Records – Pascal
- Objects – Object Pascal
- Macros – C
- Abstract Base Classes – C++
- Interfaces – Java
- Templates – C++
- Generics – Java
- Lambdas – Lisp
- Combinators – SKI Calculus
- Dataflow Graphs – 3ds Max MCG
“The effective exploitation of their powers of abstraction must be regarded as one of the most vital activities of a competent programmer.”

– Edsger W Dijkstra (1972 Turing Award Lecture)
Learning Objectives

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

Making code easier to debug, validate, extend, and modify
Software Engineering Principles

- Referential Transparency
  - Avoid side effects
  - Immutability

- Reduce Coupling
  - Dependency Injection
  - Interface Segregation

- Reduce Code
  - Don’t Repeat Yourself (DRY)
  - YAGNI
  - Robustness Principle

- Simplify
  - Single Responsibility
  - Open-Closed
  - “Simplest thing that could possibly work”
A style of programming that encourages using pure mathematical functions over procedures.

Emphasizes immutable data structures, functions as first-class values, and no side effects.

The result is that software components are easier to refactor and reason about.
What vs How

Functional programming - evaluation of mathematical expressions.

Imperative programming - series of instructions that modify internal state
Parentheses after a function name is an operator. Specifically the function application operator.
Computer Science 101 Revisited

- A value is an instance of a type.
- A type is a set of values and supported operations.
- An expression is a set of symbols (literals, identifiers, operators) that can be evaluated. Also called a “term” in type-theory.
- Evaluation is the process of reducing expressions to values.
- A function is a type of value that supports function application.
- Application is an operation on a function and arguments input values the yields an output value.
Lambda Calculus
invented by Alonzo Church in 1932

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Variable</td>
<td>A character or string representing a parameter or mathematical/logical value</td>
</tr>
<tr>
<td>(\x.M)</td>
<td>Abstraction</td>
<td>Function definition (M is a lambda term). The variable x becomes <em>bound</em> in the expression.</td>
</tr>
<tr>
<td>(M N)</td>
<td>Application</td>
<td>Applying a function to an argument. M and N are lambda terms.</td>
</tr>
</tbody>
</table>
Arithmetic in lambda calculus  [edit]

There are several possible ways to define the natural numbers in lambda calculus, but by far the most common are the Church numerals, which can be defined as follows:

\[
\begin{align*}
0 & := \lambda f.\lambda x.\ x \\
1 & := \lambda f.\lambda x.\ f\ x \\
2 & := \lambda f.\lambda x.\ f\ (f\ x) \\
3 & := \lambda f.\lambda x.\ f\ (f\ (f\ x))
\end{align*}
\]

We can define a successor function, which takes a Church numeral \( n \) and returns \( n + 1 \) by adding another application of \( f \), where \((mf)x\) means the function \( f \) is applied \( m \) times on \( x \):

\[
\text{Succ} := \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x)
\]
Luckily we don’t have to know all that

Just an interesting anecdote about why we call anonymous functions “lambdas”.
Functions are relations between values

Given a value (or set of values)
Values can be:

- Numbers (floats)
- Boolean values (True / False)
- Vectors (Cartesian coordinates in 2D and 3D)
- Geometric objects (Constructive Solid Geometry)
- Signed distance fields (function from 3D location to number)
- Vector fields (function from 3D location to 3D vector)
- Point clouds
Functions:

- Numbers (floats)
- Boolean values (True / False)
- Vectors (Cartesian coordinates in 2D and 3D)
- Geometric objects (Constructive Solid Geometry)
- Signed distance fields (function from 3D location to number)
- Vector fields (function from 3D location to 3D vector)
- Point clouds
Images are Examples of Functions

• An image is an example of a function: $f(x, y) \Rightarrow \text{color}$.
• If $x$ and $y$ are discrete values (integers) you have a raster image.
• If $x$ and $y$ are continuous (floats) you have a vector image.
More Examples of Functions

- 3ds Max Modifiers
- MAX Creation Graph (MCG)
- Maya Vector Fields
- Shader Graphs
- Parametric shapes
- NURBcs
- Mesh CSG (Boolean) Operations
- MASH (Maya Motion Graphics)
Parametric Shape

```csharp
[ExecuteAlways]
public class SampleCylinder : ProceduralMesh
{
    [Range(0, 100)] public float Radius = 10f;
    [Range(0, 100)] public float Height = 1f;
    [Range(1, 200)] public int USegments = 10;
    [Range(1, 200)] public int VSegments = 24;

    // http://mathworld.wolfram.com/Cylinder.html
    public Vector3 CylinderFunction(Vector2 uv)
    {
        return new Vector3(
            Radius * Mathf.Cos(uv.X * Constants.TwoPi),
            Radius * Mathf.Sin(uv.X * Constants.TwoPi),
            Height * uv.Y);
    }

    public override IMesh ComputeMesh()
    {
        => Primitives.QuadMesh(CylinderFunction, USegments, VSegments, true, false);
    }
}
```
The Big 3 Higher Order Functions

- Where (Filter)
- Select (Map)
- Aggregate (Fold)
You may already know higher order functions

Assuming you know SQL, there is a simple mapping between queries and higher-order functions.

```
SELECT * 
FROM Book 
WHERE price > 100.00 
ORDER BY title;
```

```
public static IEnumerable<Book> BookQuery(Book[] books) 
{
    return books 
        .Where(bk => bk.Price > 100.0) 
        .OrderBy(bk => bk.Title);
}
```
IArray – Immutable Array Interface

/// <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; }
}
Why Immutable Interfaces?

• Immutable objects can’t go wrong.
• No need for synchronizing in a concurrent situation
• Easier to reason about
• Rebinding a variable to a new value is acceptable (but is generally not a good idea)
• Implementations can evaluate properties on demand
Functional Array Implementation

```csharp
public static class FunctionalArrayDemo
{
    public static IArray<T> Create<T>(int count, Func<int, T> f)
        => new FunctionalArray<T>(count, f);

    public static IArray<T> Repeat<T>(T x, int count)
        => Create(count, i => x);

    public static IArray<int> Range(int count)
        => Create(count, i => i);

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

    public static IArray<U> Select<T, U>(this IArray<T> xs, Func<T, U> f)
        => Create(xs.Count, i => f(xs[i]));
}
"Changing" an immutable array

```csharp
public static IArray<int> ApplyChanges(
    this IArray<int> values,
    Dictionary<int, int> changes
)
{
    return values.Select((v, i) =>
        changes.ContainsKey(i)
            ? changes[i]
            : v);
}
```
Common Programming Paradigms

- Structured
- Procedural
- Functional
public class DemoUnstructured
{
    public static void Test()
    {
        var xs = new[] { 1, 2, 3, 4, 5 };

        var sum = 0;
        foreach (var x in xs)
        {
            sum += x;
        }

        var prod = 1;
        foreach (var x in xs)
        {
            prod *= x;
        }

        for (var i = 0; i < xs.Length; ++i)
        {
            xs[i] = xs[i] * xs[i];
        }

        var sum2 = 0;
        foreach (var x in xs)
        {
            sum2 += x;
        }

        var prod2 = 1;
        foreach (var x in xs)
        {
            prod2 *= x;
        }

        Console.WriteLine($"The sum and product is \{sum\} and \{prod\}"");
        Console.WriteLine($"The sum and product of squares is \{sum2\} and \{prod2\}"");
    }
}
public class DemoProcedural
{
    public static int ComputeSum(int[] xs) {
        var r = 0;
        foreach (var x in xs)
        {
            r += x;
        }
        return r;
    }

    public static int ComputeProduct(int[] xs) {
        var r = 1;
        foreach (var x in xs)
        {
            r *= x;
        }
        return r;
    }

    public static void ComputeSquares(int[] xs) {
        for (var i = 0; i < xs.Length; ++i)
        {
            xs[i] = xs[i] * xs[i];
        }
    }

    public static void Test()
    {
        var xs = new[] { 1, 2, 3, 4, 5 };
        var sum = ComputeSum(xs);
        var prod = ComputeProduct(xs);
        ComputeSquares(xs);
        var sum2 = ComputeSum(xs);
        var prod2 = ComputeProduct(xs);
        Console.WriteLine("
The sum and product is {sum} and {prod}");
        Console.WriteLine("
The sum and product of squares is {sum2} and {prod2}"");
    }
}
public class DemoFunctional
{
    public static T Aggregate<T>(T[] xs, T seed, Func<T, T> f)
    {
        foreach (var x in xs)
            seed = f(seed, x);
        return seed;
    }

    public static T[] Select<T>(T[] xs, Func<T, T> f)
    {
        var r = new T[xs.Length];
        for (var i = 0; i < xs.Length; ++i)
            r[i] = f(xs[i]);
        return r;
    }

    public static void Test()
    {
        var xs = new[] { 1, 2, 3, 4, 5 };
        var sum = Aggregate(xs, 0, (x, y) => x + y);
        var prod = Aggregate(xs, 1, (x, y) => x + y);
        var xs2 = Select(xs, x => x * x);
        var sum2 = Aggregate(xs2, 0, (x, y) => x + y);
        var prod2 = Aggregate(xs2, 1, (x, y) => x * y);

        Console.WriteLine("The sum and product is {0} and {1}.", sum, prod);
        Console.WriteLine("The sum and product of squares is {0} and {1}.", sum2, prod2);
    }
}
Functional Programming blends well with OOP

• Interfaces
  o Define an abstract data type

• Classes:
  o Either
    ▪ Implement an interface
    ▪ Container of heterogeneous data
    ▪ Helper utility (e.g. StringBuilder)

• Functions
  o Define transforms of inputs into an output
public class ClassicMesh
{
    public Vector3[] Vertices;
    public AABox Box = AABox.Empty;

    public ClassicMesh(Vector3[] vertices) {
        Vertices = vertices;
    }

    public void Translate(Vector3 offset) {
        for (var i = 0; i < Vertices.Length; ++i)
            Vertices[i] += offset;
    }

    public void Rotate(Quaternion q) {
        for (var i = 0; i < Vertices.Length; ++i)
            Vertices[i] = Vertices[i].Transform(q);
    }

    public void ComputeBox() {
        Box = new AABox(Vector3.MaxValue, Vector3.MinValue);
        for (var i = 0; i < Vertices.Length; ++i)
            Box = Box.Merge(Vertices[i]);
    }
}
public class MeshWithLinq
{
    public Vector3[] Vertices;
    public AABox Box = AABox.Empty;

    public MeshWithLinq(Vector3[] vertices)
    {
        Vertices = vertices;
    }

    public void Translate(Vector3 offset)
    {
        Vertices = Vertices.Select(v => v + offset).ToArray();
    }

    public void Rotate(Quaternion q)
    {
        Vertices = Vertices.Select(v => v.Transform(q)).ToArray();
    }

    public void ComputeBox()
    {
        Box = Vertices.Aggregate(AABox.Empty, (b, p) => b.Merge(p));
    }
}
public class MeshWithLinqFixed
{
    public Vector3[] Vertices;
    public AABox _box = AABox.Empty;

    public AABox Box {
        get {
            if (_box == AABox.Empty) ComputeBox();
            return _box;
        }
    }

    public MeshWithLinqFixed(Vector3[] vertices) {
        Vertices = vertices;
    }

    public void Translate(Vector3 offset) {
        Vertices = Vertices.Select(v => v + offset).ToArray();
    }

    public void Rotate(Quaternion q) {
        Vertices = Vertices.Select(v => v.Transform(q)).ToArray();
        _box = AABox.Empty;
    }

    public void ComputeBox() {
        _box = Vertices.Aggregate(AABox.Empty, (b, p) => b.Merge(p));
    }
}
public class ImmutableMesh
{
    public readonly IEnumerable<Vector3> Vertices;

    public ImmutableMesh(IEnumerable<Vector3> vertices)
        => Vertices = vertices;

    public ImmutableMesh Translate(Vector3 offset)
        => new ImmutableMesh(Vertices.Select(v => v + offset));

    public ImmutableMesh Rotate(Quaternion q)
        => new ImmutableMesh(Vertices.Select(v => v.Transform(q)));

    public AABB ComputeBox()
        => Vertices.Aggregate(AABB.Empty, (b, p) => b.Merge(p));
}
public class FunctionalMesh
{
    
    public readonly IEnumerable<Vector3> Vertices;

    public FunctionalMesh(IEnumerable<Vector3> vertices)
    
    => Vertices = vertices;

    public FunctionalMesh Deform(Func<Vector3, Vector3> f)
    
    => new FunctionalMesh(Vertices.Select(f));

    public FunctionalMesh Translate(Vector3 offset)
    
    => Deform(v => v + offset);

    public FunctionalMesh Rotate(Quaternion q)
    
    => Deform(v => v.Transform(q));

    public AABox ComputeBox()
    
    => Vertices.Aggregate(AABox.Empty, (b, p) => b.Merge(p));
}
Separating Implementation from Interface

```csharp
public interface IMesh
{
    IArray<Vector3> Vertices { get; }
}

public class MeshImplementation : IMesh
{
    public IArray<Vector3> Vertices { get; }
    public MeshImplementation(IArray<Vector3> vertices)
        => Vertices = vertices;
}

public static class MeshExtensions
{
    public static IMesh ToMesh(this IArray<Vector3> vertices)
        => new MeshImplementation(vertices);

    public static IMesh Deform(this IMesh m, Func<Vector3, Vector3> f)
        => m.Vertices.Select(f).ToMesh();

    public static IMesh Translate(this IMesh m, Vector3 offset)
        => m.Deform(v => v + offset);

    public static IMesh Rotate(this IMesh m, Quaternion q)
        => m.Deform(v => v.Transform(q));

    public static AABB ComputeBox(this IMesh m)
        => m.Vertices.Aggregate(AABB.Empty, (b, p) => b.Merge(p));
}
So what are the advantages?

• No shared state
• Concurrency is safe: no synchronization issues
• Once tested: functions always work (test it and forget about it)
• You can safely share data (1 bunny not 1000!)
• Code against an immutable interface
\[ x = (c + a \cos \nu) \cos u \]
\[ y = (c + a \cos \nu) \sin u \]
\[ z = a \sin \nu \]
Summary
Functional programming is an approach of expressing computation using mathematical functions instead of subroutines.
1. Shared state is the source of most bugs and complexity
1. Shared state is the source of most bugs and complexity

2. Functional programming eliminates shared state
Don’t tell computers how to compute, tell them what to compute and you will be successful!