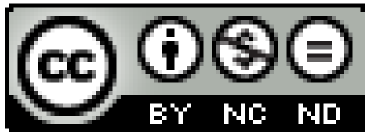
Computer Science from the Metal Up:

Case Study – Ray Tracing (VB)

By Richard Pawson

v1.0.0

©Richard Pawson, 2020. The moral right of the author has been asserted.



This document is distributed under a Creative Commons Attribution-NonCommercial-NoDervivatives 4.0 International License: https://creativecommons.org/licenses/by-nc-nd/4.0/.

The author is willing, in principle, to grant permission for distribution of derivative versions, on a case by case basis, and may be contacted as rpawson@metalup.org in relation to permissions, or to report errors found in the book.

'Metal Up' is a registered trademark, number UK00003361893.

Acknowledgements

The RayTracing code used in this workbook is based on an example created by Luke Hoban, but has been modified somewhat to suit the educational purposes of the book.

The source code for the modified version used in this workbook, both C# and VB may be found here:

https://github.com/MetalUp/RayTracing

Luke Hoban's original version (in C#) may be found here:

https://blogs.msdn.microsoft.com/lukeh/2007/04/03/a-ray-tracer-in-c3-0/

Introduction ... *1*

Part I: Object-Oriented Programming ... **2**

Object properties ... 2
Aggregation vs. Composition ... 3
Encapsulation and information hiding ... 4
Operations ... 5
Polymorphism ... 6
Inheritance ... 7
Modifying the code ... 8

Part II: Vectors ... **9**

Methods and operations ... 9
What do the vectors represent? ... 9
Dot Product ... 10
Cross Product ... 11

Part III: Functional Programming ... **12**

Conditional functions ... 12
LINQ ... 13
Defining functions as properties ... 15

Introduction

In this workbook we are going to explore an application called Ray Tracing.

Exercise 1

Start by opening the application in Visual Studio and running it.

A windows Form should pop-up on screen. Initially it will be blank, but after several seconds an image will appear within the form, showing a 3D scene comprising spheres of different sizes above a checkerboard floor.

Paste a screen snippet of the image.

The reason why the application took several seconds to respond is that it is not simply loading a ready-made image from disk (which would be virtually instantaneous); rather it is *constructing* the image, pixel by pixel, tracing rays of light, originating from multiple light sources of different colours, reflecting off various surfaces, both shiny and matt, until they pass into a simulated camera lens.

Actually, it works in reverse: each pixel within the virtual camera is traced *backwards* until it hits a solid surface, at which point it may be traced further back to another object or light source, and so on. There is a lot of quite complex mathematics involved.

Toy Story

Ray tracing is a form of computer-generated imagery (CGI) and it used extensively in the creation of animated films such as Toy Story, which typically involves creating 24 individual 'frames' per second of viewing. In the first Toy Story film (1995) *each frame* took between 45 minutes and 30 hours to render in its final form. Despite the fact that between 1995 and 2019 Moore's Law suggests that computer performance improved by a factor of 16 million, the visual techniques have become so much more sophisticated that in Toy Story 4 *each frame* took between 60 and 160 hours to render.

Despite the sophistication, the complete application you are now using is written in just over 500 lines of VB code. Unless you are willing to invest a lot of time, you might well not understand what *all* the lines are doing. But by the end of this workbook you will understand the overall structure of the program, *and be able to modify it*. This is possible because the code is written primarily using object-oriented programming (OOP) techniques , and is a good advertisement for the advantages of OOP. The code also relies heavily on the use of vectors - also implemented as objects in this case. Studying this application therefore provides useful revision on the principles of both OOP and vectors. The first two parts of the workbook deal look at those two aspects of the code, respectively.

The code also makes some use of 'functional programming' (FP) techniques, and this is explored in the third part of the workbook – as optional revision/reinforcement for those students who have covered the basic principles of FP.

Part I: Object-Oriented Programming

The Ray Tracing code consists, principally, of object class definitions.

Some of these can be considered be generic - potentially re-usable across many applications. The Vector3 class is an example of such a generic class, and we'll look at this one in Part II. Other classes, however, are custom-designed for the specific 'problem domain' of ray tracing. Typically they names of those classes correspond the nouns of that problem domain, in this case: `Scene`, `Sphere`, `Floor`, `LightSource`, `Camera`, `Ray`, and so on. Being able to see the language of the problem domain clearly reflected in the code like this makes the code easier to read, and easier to modify. This is one of the big benefits of OOP.

A class can be thought of as a 'template', or perhaps as a 'cookie cutter', from which individual instances can be constructed. A class typically defines both the properties and the methods (functions) that each instance will be created with. Another way to say this is that objects typically have both 'state' (or data) and 'behaviour' (or functionality).

Object properties

If an object class defines properties, but no methods, it is still technically an object, but it can really be thought of as data structure. Even these objects, however, will typically have a 'constructor' – which looks a bit like a method, but has a very specific purpose.

Exercise 2

What is the specific purpose of a constructor?

How is the code signature of a constructor different to other methods that might be defined on the class?

Sometimes, the constructor is defined with parameters. What is the reason for doing this?

Find an example of a class that defines properties, and a constructor, but no methods. (List the class names (no need to paste in code).

Object properties may be thought of as having two kinds: simple (value-type) properties, and associations to other domain objects. In the following example (the properties defined on the `Intersection` class), the `Dist` (distance) property is a simple value-type (`Double`), but the properties named `Thing` and `Ray` are associations to other domain objects:

```
Public ReadOnly Property Thing As Thing
Public ReadOnly Property Ray As Ray
Public ReadOnly Property Dist As Double
```

Exercise 3

What is the significance of the words `Public ReadOnly` *as used on these properties*?

Associations may also be multi-valued, in the form of an array, or list, (or any other type of collection supported by the language) of a certain domain object type.

Exercise 4

Find a class that has one or more multi-valued properties and paste in the code showing that property or properties.

Aggregation vs. Composition

Some object modellers distinguish two kinds of association: *aggregation* and *composition*. Most examples of association between objects would then be categorised as aggregation. An association may be described as composition when the associated objects may be described as 'belonging to' a single specific object, and therefore never associated with any other object. It is quite difficult to define a rigorous test for 'belonging to'. However, we can find one clear example of it in the Ray Tracing code. The `Camera` class defines four properties, each of type `Vector3`, which are set up by the constructor:

```
Public Class Camera
    Public ReadOnly Pos As Vector3
    Private ReadOnly Forward As Vector3
    Private ReadOnly Up As Vector3
    Private ReadOnly Right As Vector3

    Public Sub New(ByVal pos As Vector3, ByVal lookAt As Vector3)
        Forward = (lookAt - pos).Normalized()
        Dim down = New Vector3(0, -1, 0)
        Right = Forward.CrossProduct(down).Normalized() * 1.5
        Up = Forward.CrossProduct(Right).Normalized() * 1.5
        Me.Pos = pos
    End Sub
```

Three of those four properties (`Forward`, Up, and `Right`) are examples of Composition - they are owned by the Camera class and not referenced by any other class. We can say this because:

- The three vectors for those properties are created *within the constructor code*; they are not passed into the constructor as ready-made objects.
- The three properties in which those created objects are held, are marked `Private`. So it is not possible for another object to get hold of a reference to them and associate it with themselves.
- The three vectors are used internally by the `GetPoint` method on `Camera`, but not passed outside the object by that method.

The `Pos` property cannot be definitively categorised as 'composition' because not only is it made public, but we know that it is passed into the `Camera`'s constructor as a pre-existing object.

There are other examples within Ray Tracing that *might* be described as Composition, in that the objects are apparently associated with a single owner, but we need to be cautious because there is nothing in the code, for those other examples, that would *prevent* them from being associated with multiple other objects. For example, we might observe that the objects associated with the `defaultScene` are 'owned' by that Scene, and are only accessed (e.g. via the Renderer) via that `Scene`. But this is not necessarily always true. We can easily define further `Scenes` e.g. within (the plural name even hints at this possibility) referencing some of the same objects as set up for the `defaultScene`, together with some different ones.

Encapsulation and information hiding

Object classes may also define instance methods - such that each instance of that class will have (or, at least, appear to have) a copy of those methods, but applying to their own individual data (state).

Like properties, instance methods may be marked as `Public` or `Private` - where private methods may be called only from inside the object.

Some methods are marked up with the `Shared` keyword. These are *not* instance methods. They are really like free-standing functions, that are just associated in some way with a class for convenience.

Exercise 5

Find an example an instance method. Write the name of the instance method together with the name of the class on which it is defined.

By right clicking on the name of the method and selecting **Find all References** find an example of the instance method being called from elsewhere in the code. Paste in the line of code showing where that is being called, and highlight or state separately the name of the instance (e.g. variable name or parameter name) that it is being called on.

(C#) or (VB) When a class definition includes instance methods we say that those methods are 'encapsulated' on the object. What are the advantages of doing this?

- Packaging a data structure together with the common functions that operate on that data structure can make it easier to re-use the code between different applications. However, this is not unique to OOP - there are other ways to package up functions and data structures into re-usable components.
- The 'dot syntax'. An instance method is invoked on an object instance, by typing a '.' And then the method name. In fact, when you type a '.' after an object reference (such as a variable name) the IDE will typically offer the programmer a list of the methods that can be invoked on that type, which is very convenient. Note that, where a method returns an object, it is also possible to 'chain' method calls, using the dot-syntax, where each method call is applied to the result returned by the previous one (which might be of a different type to the original one). Again, however, although commonly associated with OOP, this capability is not unique to OOP. (If you want to know more, search the web for VB 'extension methods').
- An advantage of encapsulation that is unique to OOP is known as 'information hiding'. This is where an object defines some, or all, of its properties as private, but reveals public methods that use that private information.

The principle of 'information hiding' is well illustrated in the `Colour` class, which defines three `private` properties of type `Double`:

```
Public Class Colour
    Private R, G, B As Double

    Public Sub New(ByVal r As Double, ByVal g As Double, ByVal b As Double)
        Me.R = r
        Me.G = g
        Me.B = b
    End Sub
```

It is more common to use integers to represent the RGB (Red, Green, Blue) components of a colour. Ray Tracing uses `Double` because it performs a lot of mathematical manipulations on the colours and treating them as `Double`s results in less loss of precision. However, to allow the colours to be displayed on the screen, `Colour` provides public methods that returns each of the RGB values as a byte (a byte being the same as an integer in the range 0-255).

```
Private Function ToByte(ByVal d As Double) As Byte
    Return Convert.ToByte(Math.Min(1, d) * 255)
End Function

Public Function RedByte() As Byte
    Return ToByte(R)
End Function

Public Function GreenByte() As Byte
    Return ToByte(G)
End Function

Public Function BlueByte() As Byte
    Return ToByte(B)
End Function
```

Calling such a method does lose colour precision, but this is done only *after* all the calculation involving colour values has been completed.

Operations

While we are talking about manipulating colours, the Colour class defines instance methods: `Plus`, `Minus`, and `MultiplyBy`. The idea of adding and subtracting colours is easy to understand, but what does it mean to multiply colours? In fact, there are two versions of `MultiplyBy` method, both having the same name, but taking different parameters. We say that the `MultiplyBy` method is 'overloaded'.

Exercise 6

Find the two versions of the `MultiplyBy` method on Colour, and state, in each case, what parameters it takes, and what it returns.

Looking at their implementations, describe what each is doing in English.

The effect of multiplying a colour by a number is to increase (or decrease) the brightness of the colour, while keeping the same colour balance. Multiplying a colour by another colour is used to calculate the colour that is reflected from a surface: if a pure red light is shone on a pure red surface, it will reflect pure red, but if a pure blue light is shone on a pure red surface, nothing will be reflected. Most everyday colours are mixtures of the three primaries, which is why a yellow light, shone on a purple surface will reflect as red.

These methods could be called extensively within the Ray Tracing code, but they aren't. Because `Colour` also defines 'operators' – each using the operator keyword – in a region at the bottom of the class:

```
Public Shared Operator *(ByVal n As Double, ByVal c1 As Colour) As Colour
    Return c1.MultiplyBy(n)
End Operator

Public Shared Operator *(ByVal c1 As Colour, ByVal c2 As Colour) As Colour
    Return c1.MultiplyBy(c2)
End Operator

Public Shared Operator +(ByVal c1 As Colour, ByVal c2 As Colour) As Colour
    Return c1.Plus(c2)
End Operator

Public Shared Operator -(ByVal c1 As Colour, ByVal c2 As Colour) As Colour
    Return c1.Minus(c2)
End Operator
```

As you can see, the operator definitions look like functions except that their 'names' are the mathematical symbols that we used to perform arithmetic on numbers - and they work the same way. Note that:

- Each operator takes exactly two parameters
- Operator definitions must be declared as `Shared`.
- The implementation of each operator just delegates to the appropriate instance method on one of the parameters

The nice thing about operators is that instead of calling the function like this:

```
+(a,b)
```

It is called like this:

```
a + b
```

Exercise 7

What is the correct name, in computer science, for the difference between the two forms of notation for an operator shown immediately above?

The following method (from the `Renderer` class) shows the multiply operator (for `Colour`) in use:

```
Private Function GetReflectionColor(ByVal thing As Thing, ByVal pos As Vector3,
              ByVal norm As Vector3, ByVal rd As Vector3, ByVal scene As Scene,
              ByVal depth As Integer) As Colour
    Return thing.Surface.Reflect(pos) * TraceRay(New Ray(pos, rd), scene, depth + 1)
End Function
```

Exercise 8

Re-write the implementation above without using the operator – by calling method(s) on the colour objects instead.

Polymorphism

In the previous section we looked at some advantages of encapsulation, but now we come to the most important one: *polymorphism*. In OOP, polymorphism is where two or more object classes define a member with the same signature, even though the implementation may be quite different. For example, both `Floor` and `Sphere` define a method with this signature:

```
Public Overrides Function CalculateIntersection(ByVal withRay As Ray) As Intersection
```

If you look at the method body (the implementation) on the two objects you will see that it is quite different.

Now because the implementation is encapsulated with the object, it means that when we want to calculate the intersection of a ray with a physical object that might be a `Floor` or might be a `Sphere`, we don't need to implement an `if` statement or a `Select…Case` statement to select the appropriate implementation of that function, we can just call the `CalculateIntersection` method on the object itself.

In fact, we could have a list of the physical objects in the scene and call that method on each in turn, *without even needing to know which specific type of object we are dealing with in each case*.

Polymorphism only works if the programming language has a mechanism to support it. C#VB has more than one such mechanism. The simplest one is *inheritance*.

Inheritance

Now look at the function named `DefaultThings` on the `StandardScenes` class and notice that it returns an array of type `Thing`, which has four members: one `Floor`, and three `Spheres`.

Exercise 9

How is the array able to hold two different types of object? (Hint: look at the definitions of the classes `Sphere` and `Floor`.)

`Thing` defines two instance methods – `CalculateIntersection` and `CalculateNormal`, but neither of them contains a 'method body' i.e. code to execute when the method is called. How/why is that possible? Will any code be executed when either of those methods is called on any of the `Things` in the array returned by `DefaultThings`?

Name another sub-class of `Tbing` that you could imagine being written for the application (don't worry - you won't be asked to write it as part of this workbook.)

In addition to being a simple way to enable polymorphism, inheritance also offers the option of providing a standard implementation of a member (a member can be a property or a method, remember) to be used, automatically, by any sub-class that doesn't provide its own specific implementation.

Exercise 10

`Sphere` defines two properties, but also inherits one property. What are the names and types of all three properties that are available on `Sphere`.

Modifying the code

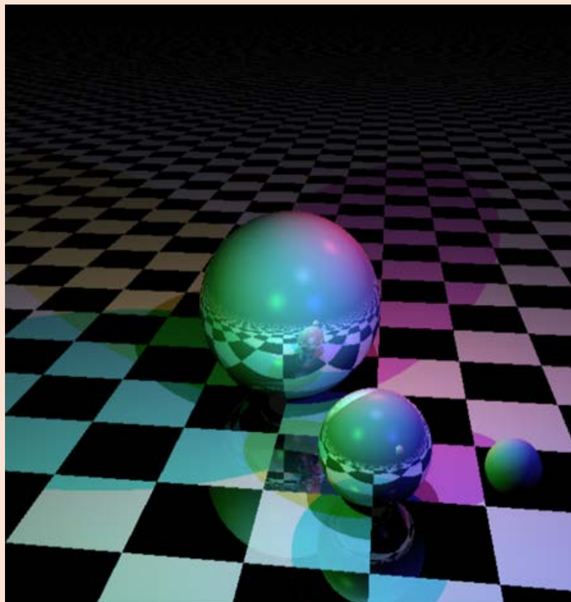
Exercise 11

Within `StandardScenes` you will find the `DefaultCamera` function which creates and returns a single instance of `Camera`.

Your task is to create a new view of the same scene, by editing both the position and viewpoint of that `Camera` instance. You'll need to figure out, by looking at the code, which of the two `Vector3` parameters passed into the Camera's constructor defines its position, and which defines where the camera is pointing, and then you'll need to figure out which is the x,y, and z coordinate for the Vector, and what those represent in terms of the image you are looking it.

Hint: The best way to do this is to make *small* adjustments to the values, and to modify just one value at a time, restoring the previous value between experiments until you are confident as to their meaning or effect, which will allow you to make larger changes in one go.

Modify just this function to achieve a view of the scene like the one below. Your view does not have to be *identical* to one shown here, but it should be broadly similar - with the largest sphere in the centre of the image, the other two being nearer the camera, and the whole scene viewed from a steeper angle.



Paste in a screen snippet of your rendered image, and your modified version of the `DefaultCamera` function.

Exercise 12

Keeping the same view add another medium-sized sphere to the `DefaultScene`, somewhere where it is in view and without *completely* hiding the existing three (it may *partly* obscure them). The new sphere should be *shiny*. Also make the floor matt instead of checkerboard.

Paste a screen snippet showing your rendered scene and your new code for the `DefaultScene` function.

Part II: Vectors

You will already have observed that the `Vector3` class is used throughout the Ray Tracing code. You can verify this by clicking on any use of `Vector3` and selecting **Find All References.**

If you were asked to define a vector, you *might* say something like this: 'a vector represents a line with both length and direction'. This would be true of *geometric* vectors (also known as Euclidean vectors), both 2D and 3D, but it is important to understand that in Computer Science, not all vectors are geometric. A vector is really a data structure containing a defined number of numeric elements of the same type (typically integers or real numbers) with the elements held in fixed positions. Thus, it is perfectly possible to have a vector with 4, 5 or 100 elements.

Does this sound just the same as an array, or a dictionary, perhaps? It is. So what makes something a vector. The answer is simply that it is how you use it that makes it a vector, rather than an intrinsic property. In other words an array of numbers is a vector, *if we perform vector operations on it*, for example, vector addition, vector subtraction, dot product, modulus, and so on. Vector operations state how the various elements of the one, or two, input vectors are combined to form the result.

Although you don't need OOP to implement these vector operations, if you are going to do a lot of vector operations it makes a lot of sense to encapsulate the vector operations with the vector data - and that's what we have in the `Vector3` class – a simple implementation of a 3-dimensional vector.

Methods and operations

Exercise 13

Go to the `Vector3` class definition. What is the type of each of the three properties that it defines?

How would this vector be defined using the formal mathematical notation?

List the names of the *instance* methods that `Vector3` defines.

In addition, Vector3 defines three operators (which we learned about in Part I on the `Colour` object), one of which is overloaded. Which one?

Which of these methods and operators modify an existing vector, and which create a brand new `Vector3` as the result?

Describe, in words, how the resulting `Vector3` returned by the Normalized method differs from the `Vector3` that it is called upon?

You might be puzzled by the use of a 'vector' to define a point in space. Surely a vector represents a line with a length and a direction? Remember that a vector is fundamentally just a data structure that holds a fixed number of values – in this case, three. It *can* be used to represent a line with length and direction, or a point in 3D space, or many other things. The application *could* have been written using arrays of size three to represent these things, instead of a special `Vector3` class, but we'll look at the advantages of using a special `Vector3` class in the second part of this workbook.

What do the vectors represent?

The vectors used in the Ray Tracing code are all three-dimensional *geometric* vectors, but many of them don't appear to fit the simple description of vector as being 'a line with length and direction'. Some of them seem to represent a point in 3D space – for example the location of the centre of a Sphere – while others seem to represent just a direction – for example the vector representing the

'normal' direction of a surface ('normal' meaning the direction that is at right angles to the surface at that point).

This can become clearer if we look at the Ray class, which represents a ray of light:

```
Public Class Ray

    Public ReadOnly Property Start As Vector3

    Public ReadOnly Property Dir As Vector3

    Public Sub New(ByVal start As Vector3, ByVal dir As Vector3)
        Me.Start = start
        Me.Dir = dir.Normalized()
    End Sub
End Class
```

We can see that the Ray comprises two vectors, one representing the point in space where it originates (`Start`), and one representing the direction in which it is pointing (`Dir`). Both must be provided to the constructor when creating a new `Ray`. Notice also that the vector representing direction is 'normalized' to set it to a standard length of 1. This probably isn't strictly necessary, but it is a good way to make it clear that we are interested only in the direction that this vector defines, not its length. The calculations that use the Ray will extend its length as far as it is needed for the application.

Exercise 14

How could you extend the length of any Vector3 without changing its direction?

Dot Product

`Vector3` defines a `DotProduct` instance method, which takes another Vector3 as a parameter and returns a 'scalar' value (a `Double`). It is used in several different places within the code.

Exercise 15

Describe in words, how the result returned by `DotProduct` is calculated (either from memory, or looking at the implementation).

If the two vectors are geometric (either 2D or 3D) what is another way of describing the *significance* (in geometric terms) of their dot product?

An interesting example of how this is used may be seen in the `CalculateIntersection` method of a `Sphere`:

```
Public Overrides Function CalculateIntersection(ByVal withRay As Ray) As Intersection
    Dim eo As Vector3 = Centre - withRay.Start
    Dim v As Double = eo.DotProduct(withRay.Dir)
    Dim dist As Double

    If v < 0 Then
        dist = 0
    Else
        Dim disc As Double = Math.Pow(Radius, 2) - (eo.DotProduct(eo) - Math.Pow(v,
2))
        dist = If(disc < 0, 0, v - Math.Sqrt(disc))
    End If

    If dist = 0 Then Return Nothing
    Return New Intersection(Me, withRay, dist)
End Function
```

- The first line calculates a new vector that represents the direction from the centre (point) of the sphere to the start (point) of the ray, by subtracting the latter from the former.
- The second line calculates the dot product of this with the direction of the ray
- If this dot product (v in the code) is less than zero it means that the ray is point away from the sphere, so the method returns `Nothing`.
- If v is greater than zero then a further calculation is performed, starting by checking if the ray passes within one radius of the centre of the sphere…

Cross Product

The cross product of two vectors results in another vector. For a three-dimensional geometric/Euclidean vector the resulting vector will always point in a direction that is at right angles to both the input vectors. It's length will be proportionate to the area defined by the parallelogram defined by the other two vectors.

The `CrossProduct` instance method on Vector3 is used only within the constructor of the `Camera` class:

```
Public Sub New(ByVal pos As Vector3, ByVal lookAt As Vector3)
    Forward = (lookAt - pos).Normalized()
    Dim down = New Vector3(0, -1, 0)
    Right = Forward.CrossProduct(down).Normalized() * 1.5
    Up = Forward.CrossProduct(Right).Normalized() * 1.5
    Me.Pos = pos
End Sub
```

- The first line of this code calculates a new vector representing the direction in which the camera is pointing (by subtracting the point it is looking at from the point of its position). The call to `Normalized` helps to clarify that we are interested only in the direction, not the distance.
- In the second line a new vector `down` is defined in absolute coordinates (i.e. aligned to the y axis of the scene).
- Cross product is then used to create vectors that point to the `right`, *relative to the view of the camera*, and then upwards, *relative to the view of the camera*.

Part III: Functional Programming

Although Ray Tracing is written primarily using the object-oriented programming (OOP) paradigm, it does make use of some functional programming (FP) techniques within the objects. It is thus a good example of 'multi-paradigm programming'. Many modern programming languages (C#, VB, Python, for example) support multi-paradigm programming. The advantage of this approach is that you can 'cherry pick' the best features of each approach, or apply the techniques where they are most effective. However, it can be argued that you fail to gain the *full* advantage of either one of them. We'll not pursue that debate here, but we will use the opportunity to revise/reinforce *some of* the principles of FP by looking at where they are used in Ray Tracing.

Conditional functions

In procedural programming and OOP you will be used to the idea of 'if statements' or 'selection'. However, RayTracing also makes use of the 'conditional function', which is also known as the 'ternary operator'. Shown below is an example from both the C# and the VB version of the program:

C#

```
Colour lcolor = illum > 0 ? illum * light.Color : new Colour(0, 0, 0);
```

VB:

```
Dim lcolor As Colour = If(illum > 0, illum * light.Color, New Colour(0, 0, 0))
```

These two code snippets are directly equivalent. The conditional function or ternary operator in C# involves a `?` and a `:`. It is more terse than the VB equivalent, but it can be argued that the meaning is clearer in the VB version, which uses the '`If` operator' - not the same as an ordinary '`If` *statement'*.

Using a regular `If` *statement*, the code above would look like this:

C#:

```
Colour lcolor = null;
if (illum > 0)
{
    lcolor = illum * light.Color;
}
else
{
    lcolor = new Colour(0, 0, 0);
}
```

VB:

```
Dim lcolor As Colour
If illum > 0 Then
    lcolor = illum * light.Color
Else
    lcolor = New Colour(0, 0, 0)
End If
```

The conditional/ternary *operator* is more succinct, but there is another important distinction. Again, it is perhaps clearer in the VB example. The `If` *operator* returns a value, which may be assigned to a variable (`lcolor` in this case). The value that is returned depends upon whether the condition evaluates to `True` or `False`. An `If` *statement*, does not return a value - it transfers the execution to one of two different locations in code according to the evaluation of the condition.

The VB syntax also makes it clearer that the conditional/ternary operator is really a function, and that it is sometimes called the 'ternary operator' because the function takes three arguments: the condition, the value to return if the condition is true, and the value to return if the condition is false. (Each of the two values may be expressions that may be evaluated to generate the return value.)

Spreadsheets have an IF function

If the idea of 'If as a function' sounds vaguely familiar, perhaps you have used the `IF` function on a spreadsheet. This, too, has three arguments: a condition, and two expressions, which may be as simple as references to two other cells, one yields the value if the condition evaluates to `true`, the other if the condition evaluates to `false`.

A2 =IF(A1>5, B1, C1)

	A	B	C	D	E
1	7	Big	Small		
2	Big				
3					

LINQ

Ray Tracing makes use of LINQ (Language Integrated Queries) within this method (on the `Scene` class):

```
Public Function IntersectionsWith(ByVal ray As Ray) As List(Of Intersection)
    Return Things.Select(Function(t) t.CalculateIntersection(ray)).
        Where(Function(inter) inter IsNot Nothing).
        OrderBy(Function(inter) inter.Dist).ToList()
End Function
```

(Note: the body of this method is a single statement, but it has been formatted over three lines for readability).

This method generates a list of all the intersections between a `Ray` (passed in as a parameter) and all the `Things` in the `Scene`. It makes use of the `CalculateIntersection` method on `Ray`, calling it for each `Thing` in the scene. It also eliminates any `Nothing` results from those calls (i.e. where the `Ray` never touches the `Thing`) and, before returning the list of `Intersections`, sorts them by the distance from the source of the `Ray` to the point of intersection on the `Thing`, nearest first.

Writing that same method, without using LINQ would take considerably more code. For example:

```
Public Function IntersectionsWith(ByVal ray As Ray) As List(Of Intersection)
    Dim results = New List(Of Intersection)()

    For Each thing In Things
        Dim intersect = thing.CalculateIntersection(ray)

        If intersect IsNot Nothing Then
            Dim index As Integer = 0

            While results.Count() > index AndAlso intersect.Dist > results(index).Dist
                index += 1
            End While
            results.Insert(index, intersect)
        End If
    Next

    Return results
End Function
```

LINQ is a functional programming pattern. `Select`, `Where`, and `OrderBy` are all functions that take in another function as a parameter.

Exercise 16

In Functional Programming, what is the name given to a function that takes another function as a parameter?

So, looking at the first example, `Select`:

```
Things.Select(Function(t) t.CalculateIntersection(ray))
```

We can articulate this code as 'for each `Thing`, `t`, in `Things`, return the result of calling `t.CalculateIntersection(ray)`'. The result of this expression will be a collection of `Intersections`. In other words, `Select` is applied to a list of one type, and returns a list of the same length where each element is derived in some way from an element in the input list. The elements might be of the same type, or of a different type, depending on the function being applied.

Looking at the second example, `Where`:

```
Where(Function(inter) inter IsNot Nothing)
```

We can articulate this as 'for each `Intersection`, `inter`, in the collection to which this is being applied (the result of the previous operation, above), include that `inter` only if its value is not `Nothing`. So `Where` returns a collection that is a sub-set of the collection it was applied to.

Exercise 17

You should recall that functional programming has three general-purpose, higher-order functions called Map, `Filter`, and Reduce (sometimes referred to as Map, `Filter`, `Fold`).

Using your understanding of Map, `Filter`, Reduce/`Fold`, to state which of these the two LINQ functions, `Select` and `Where,` correspond to.

`OrderBy` is another high-level function, which sorts the results, in this case by the value of the `Dist` property on each .

In all three cases, *the functions being passed in* to the higher-order functions, are defined 'inline' i.e. exactly where it is needed, inside the brackets. Unlike a regular functions defined elsewhere in the code, this inline function is not given a name - we say it is an 'anonymous function'. Another term for an inline, anonymous function is a 'lambda'.

It is also possible to pass a function that has been defined elsewhere in code, and even to hold a function in a property.

Defining functions as properties

The SurfaceTexture defines the following properties:

```
Public ReadOnly Property Diffuse As Func(Of Vector3, Colour)
Public ReadOnly Property Specular As Func(Of Vector3, Colour)
Public ReadOnly Property Reflect As Func(Of Vector3, Double)
Public ReadOnly Property Roughness As Double
```

The last property is of type Double, but the first three properties each hold a function (indicated by the Funckeyword). The code further specifies that the signature of the functions in the first two properties (Diffuse, Specular) must take in a single parameter of type Vector3 and return a result of type Colour. The function held in the Reflect property must take in a Vector3 and return a Double. Between them these functions define whether/how a ray hitting the surface will be reflected and/or diffused.

You can also see that the value of these read-only properties are set up in the SurfaceTexture's constructor, into which they are passed as parameters.

Exercise 18

Using **Find All References** on the SurfaceTexture's constructor, find the place in code where instances of SurfaceTexture are being created. How are the functions that the constructor requires being specified there?

Several of the functions being specified are very simple: irrespective of the vector (pos) that they are passed, they return a fixed value for the colour, or, in the case of the function that defines the 'reflectivity' of the surface (Reflect), just return a constant number.

However, there are two examples where the function does something slightly more complex, involving a conditional/ternary operator. Where is this, and can you explain, in words, why these involve some conditional logic (you don't need to explain *exactly* how the function operates).

Can you think of an example of a different surface texture that it might be possible to write that would, similarly, require a function where the result varies according to the value passed in.

Having set up each SurfaceTexture with the functions that it needs, let's look at how are these functions used within the rest of the code.

Exercise 19

Going back to the SurfaceTexture class, invoke **Find All References** on the Diffuse property, and navigate to the reference in the Renderer class.

Paste the line of code this points to.

Describe in words how the function in the surface's Diffuse property is being used?