Advanced Programming

- Julien Vulliet (Vux)
- Elias (Elias)

Advanced Programming

- Introduction
- Snippets
- OOP
- Threading
- Pointers
- Data Structures/Custom Types
- Windows Interop
- File Access
- SlimDX

Advanced Programming

Introduction

Advanced Programming

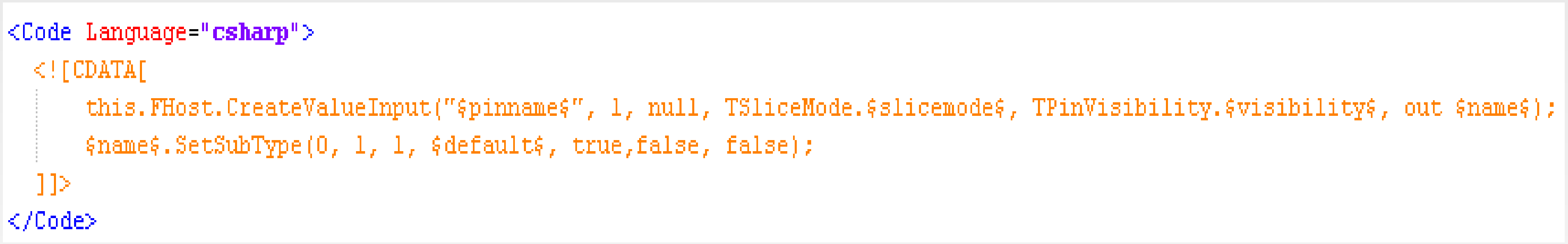
Code Snippets

A code snippet is a short programming construct.

Allows to encapsulate some long coding tasks (For example, creating Pins)

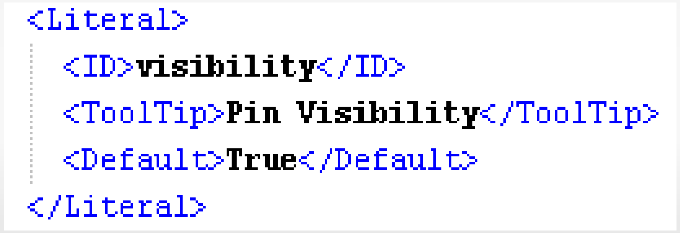
They are built as XML files.

Example (Code Section):



```
<Code Language="csharp">
  <![CDATA[
      this.FHost.CreateValueInput("$pinname$", 1, null, TSliceMode.$slicemode$, TPinVisibility.$visibility$, out $name$);
      $name$.SetSubType(0, 1, 1, $default$, true,false, false);
  ]]>
</Code>
```

Creating the variables list is done like this:



```
<Literal>
  <ID>visibility</ID>
  <ToolTip>Pin Visibility</ToolTip>
  <Default>True</Default>
</Literal>
```

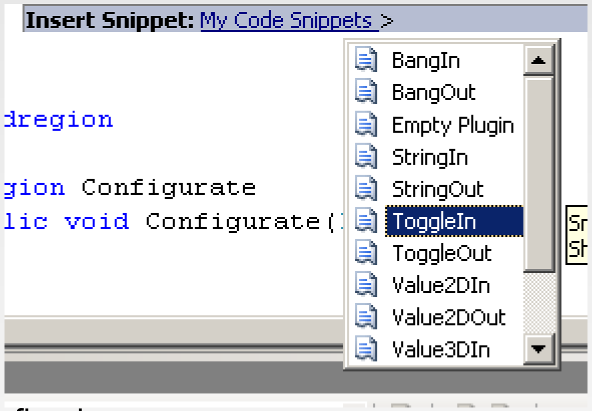
Advanced Programming

Code Snippets

To use a code snippet, we just need to copy the snippets files in:

My Documents/Visual Studio 2005/Code Snippets/Visual C#/My Code Snippets

Then we type CTRL+K → CTRL+X to display the list:



```
dregion

gion Configurate
lic void Configurate(
```

Then we can tab trough the variables we defined.

Small Exercise:

- Modify the Empty plugin snippet to add your name as Author, and add a variable for Tags
- Create a ValueIntIn/ValueIntOut snippet

Advanced Programming

OOP

Object oriented programming is a paradigm where we use objects to design applications.

We provide abstractions such as classes to design our application state and behaviour.

Most modern languages supports OOP, alothough implementations might differ completely.

The common language features are:

- Classes/Objects
- Interfaces
- Inheritance (abstract classes)
- Polymorphism (virtual methods)

Advanced Programming

Classes

A class is a data structure where we define it's state (fields) and behaviour (methods). Any element of the structure can be protected in different ways, so we can protect an object's internal state to be accessed from outside.

As a simple example we gonna define some simple geometric shapes.

Let's define a box first. It has a center, a width and a height (we leave rotation out for the moment).

```
public class Box
{
        private Vector2D center;
        private Vector2D size;

        public Vector2D Center
        {
                get { return this.center; }
                set { this.center = value; }
        }

        public Vector2D Size
        {
                get { return this.size; }
                set { this.size = value; }
        }

}
```

Classes

Now we have our class definition, we want to be able to know our Box area, we also want to be able to move our box.

```
public double Area
{
        get { return this.size.x * this.size.y; }
}

public void Move(Vector2D offset)
{
        this.center += offset;
}
```

To create a box object (also called an instance of the Box class) we simply do:

```
Box box = new Box();
```

As we noticed we have different keywords to declare our methods and fields, they are called protection level.

- Private: this protection level means our field/method is only available from within the class itself and not to the outside.
- Protected: our field/method is only available from within the class and any subclass (which we will cover later)
- Public: our field/method is also accessible from outside the class.
- Internal (not used here) : this is a special keyword that allows our field/method to be publicly accessed from any other class within the same assembly. This is generally useful when programming some APIs.

Classes

Now we have a nice Box, we gonna add a circle. It behaves the same way, the only difference is we need a radius instead of a size, and area is calculated differently.

```
public class Circle
{
        private Vector2D center;
        private double radius;
        public Vector2D Center
        {
                get { return this.center; }
                set { this.center = value; }
        }
        public double Radius
        {
                get { return this.radius; }
                set { this.radius = value; }
        }
        public double Area
        {
                get { return this.radius * this.radius * Math.PI; }
        }
        public void Move(Vector2D offset)
        {
                this.center += offset;
        }

}
```

Advanced Programming

Inheritance

Now we have our box and circle class, we can notice that they share some common data (center), some common action (Move, which is exactly the same, and Area, which differs in the implementation).

Both Box and Circle are Geometric Shapes, so we gonna create a Shape base class for it.

```
public abstract class Abstract2DShape
{
        protected Vector2D center;

        public abstract double Area { get; }

        public Vector2D Center
        {
                get { return this.center; }
                set { this.center = value; }
        }

        public void Move(Vector2D offset)
        {
                this.center += offset;
        }
}
```

Our center is now protected, we can access it's inernal state in the subclasses.

Area is now marked a abstract, as our shape doesn't know (yet) how to calculate it.

As we have one abstract method/property, our class has to be specified as abstract. We can instanciate a shape directly as some parts have no implementation.

Inheritance

Now we have our base shape definition, let's rewrite our box and circle classes:

Our box now becomes:

```
public class Box : Abstract2DShape
{
        private Vector2D size;

        public Vector2D Size
        {
                get { return this.size; }
                set { this.size = value; }
        }

        public override double Area
        {
                get { return this.size.x * this.size.y; }
        }
}
```

Box now inherits from the shape center. We also need to implement the Area property. As it comes from a parent class, we need to specify we want to override it.

Interfaces

Interfaces are another construct, which defines behaviour ONLY. An interface has no state.

This can be seen as a ”contract”, we provide a behaviour without giving any implementation specifics.

So let's say we want to be able to do hit tests on geometrics shapes. We want to be able to Test a point and see if it's inside our shape.

```
public interface I2dShapeHitTester
{
        bool TestPoint(Vector2D point);
        bool TestPoint(Vector2D point, Matrix4x4 transform);
}
```

The first method just checks the point.

The second method provides a transform so we can move the object.

Then we can create a class that implements this interface.

Some notes:

- As an interface is a contract that is supposed to be called from outside, there is no protection on methods. All interface methods have to be declared as public.
- A class can implement multiple interfaces, but can't inherit from more than one class (in c#).

Advanced Programming

Interfaces

Now we can implement our interface in our shape definition:

```
public abstract class Abstract2DShape : IShape2dHitTester
```

Now we need to provide an implementation for testing points.

As it depends on the shape, it will need to be abstract:

```
public abstract bool TestPoint(Vector2D point, Matrix4x4 transform);
```

The other method can be calculated from this one by providing an Identity Matrix:

```
public bool TestPoint(Vector2D point)
{
        return this.TestPoint(point, Vmath.IdentityMatrix);
}
```

Now we also need to implement the Hittest within our Box/Circle classes.

Advanced Programming

Generics

C# Generics is a feature to allow to define type safe data structures, without specifying a specific data type.

Previously when creating generic containers we had to rely either on using System.Object (which means we can have casting errors), or create an implementation for every single type.

Generics provides a much nicer way to handle these cases.

For example all c# containers (List,LinkedList,Stack) are implemented using generics.

Let's see how we can declare a generic sorting class

```
public class Sorter<T>
{
        public bool Reverse { get; set; }
        public List<T> Sort(List<T> input)
        {
                List<T> result = new List<T>(input);
                result.Sort();
                if (this.Reverse) { result.Reverse(); }
                return result;
        }
}
```

Now we can simply declare a subclass specific to double datatypes:

```
public class DoubleSorter : Sorter<double> { }
```

Or we can instanciate directly:

```
Sorter<double> dblsorter = new Sorter<double>();
```

Exercise: Generic Stack Node

As a first exercise, we will build a node which implements a stack. We will create it using generics, so it can implement value/string/color... efficiently and all in a uniform way.

A stack has 3 main methods:

- Push : push one item on top of the stack
- Pop : removes and return the item on top of the stack
- Peek : gets the item on top of the stack without removing it

A base implementation for stack is available in the materials folder:

The following features needs to be implemented:

- Override the class so we have a value/string/color and transform version.
- There is a potential bug on the pop section.
- Add and extra output for the Peek feature (Show item on top of the stack without removing it).
- Make the inputs/outputs spreadable, so we can push/pop/peek a spread instead of a single item.
- Add a maximum stack size input and make sure that we can't push an item if the stack is full (a stack size < 0 will mean infinite stack size). Also make sure to handle the fact that the stack size might change.

Delegates

A c# delegate is a specific datatype that references a method instead of data. It can be compared to c style function pointers in some extent.

A c# delegate needs to have a signature.

Some general uses for delegates are:

- Callbacks
- Events
- Passing custom algorithms as parameters

For example, we can provide an error delegate template:

```
public delegate void ErrorDelegate(int code, string message);
```

In our case we will look mainly how to use delegates in events contexts.

Events

C# events is a way to allow different classes to communicate without the need to introduce too much dependencies.

An event is created using delegates.

As an example let's have a class which process something and either send a success/error event.

First let's create our two delegates:

```
public delegate void SuccessDelegate();
public delegate void ErrorDelegate(int code, string message);
```

Now let's create our class which will send the events:

```
public class EventSenderExample
{
        public event ErrorDelegate Error;
        public event SuccessDelegate Success;
}
```

To raise event, we then only need the following code:

```
if (this.Error != null)
{
        this.Error(code, message);
}
```

We need to check in the case the event is null (eg: no other class is listening to it).

Events

Now we have our class which sends event, we need to create a listener.

```
public class EventListenerExample
{
        private EventSenderExample sender;
        public EventListenerExample()
        {
                sender = new EventSenderExample();
                sender.Success += new SuccessDelegate(sender_Success);
                sender.Error += new ErrorDelegate(sender_Error);
        }
        private void sender_Success()
        {
                //Raised in case of success
        }
        private void sender_Error(int code, string message)
        {
                //Raised in case of error
        }
}
```

Advanced Programming

Threading

A thread is a processing unit that computers can run concurrently.

A single process can run multiple Threads in parallel. Multiple threads access the same memory from a process.

For example: VVVV runs it's own main thread for evaluating/rendering, but Directshow nodes run on their own threads, which are synchronized at the end.

Examples of use:

- Listen to a device (Passive background task)
- Execute a lenghty task in the background (Send a file through FTP), then notify of the progress/result
- Split tasks in multiple units (to use multiple processor capabilities)

Threading

To create a thread, we need to do the following:

Import threading namespaces :

```
using System.Threading;
```

Creating a thread is then done like this:

```
 Thread thr = new Thread(new ThreadStart(MyThreadedMethod));
 thr.Start();
```

Generally we will want to encapsulate threaded code in a class, to handle the generic behaviour correctly.

Background Threads

A background thread is generally passive. It ”listens” for some incoming data (network, device), then notifies when it receives it (usually using delegate/event)

Our class will contain the following default fields:

```
 public class PassiveThreadTemplate
{        //This is our thread
         private Thread thr;
         //Indicates if our thread is running
        private bool m_running;
}
```

We will then implement the 3 following methods:

- Run (private) : Generally an infinite loop which will wait for incoming data, and raise events when it happens.
- Start (public) : Creates the thread to execute the Run method in the background
- Stop (public) : Stops processing

Advanced Programming

Send events from Threads

Sending events fromThreads is not much different from sending event from any class, apart we obviously need to take into account that code can now run concurrently.

We simply need to create a delegate, then we can fire the event.

```
public delegate void ThreadTaskCompleted();
```

Then in our Thread class we add the following methods:

```
public event ThreadTaskCompleted Completed;
 protected void OnComplete()
{
        if (this.Completed != null)
        {
                this.Completed();
        }
 }
```

Note : we generally create a protected method to fire the event, so if we fire it from multiple places in the code it makes it easier to handle specific cases

Advanced Programming

Join Threads

Joining a thread will block the caller until the thread terminates. We can use this to split a task into multiple parts.

We generally have two main uses cases for this feature:

- Data parallelization : when we have a big list to process in a uniform way, we can split this list into multiple parts and assign a thread for each part (for example, with a 10000 elements, we can process it in two chunck of 5000)
- Task parallelization : We can have two different tasks to process at once, when each other is independent. We can process both at once, then wait for both of those to be completed (for example set a velocity pin and a position pin in a particle system).

Please note that creating threads has a cost, so doing multiple cores processing will not necessarily be faster.

For example, if we want to perfom 100 additions, by the time we create a thread we would already have done the processing.

Another option is to use the parallel construct.

Lock (Basics)

Concurrent programming introduces extra complexity, due to the fact that some data might be accessed at the same time. For example we might be reading trough a list while a thread is writing into it.

For example, let's say we have a thread which populates a list, then we want to output our list content in the evaluate part of a plugin:

```
public class LockingExample
{
        private List<double> buffer = new List<double>();

        public void OnDataReceived(double data)
        {
                buffer.Add(data);
        }
        public void OutputData(IValueOut pin)
        {
                pin.SliceCount = buffer.Count;
                for (int i = 0; i < buffer.Count; i++)
                {
                        pin.SetValue(i, buffer[i]);
                }
                buffer.Clear();
        }
}
```

Advanced Programming

Lock (Basics)

Now what happens if while when in the middle of ”OutputData”, we receive an event OnDataReceived.
The data will while in the buffer, and on the next access, we will have an error.

To avoid that, we need to make sure that OnDataReceived can't be called at the same time as OutputData. We use a lock to achieve this.

```
private object m_lock = new object();
```

then our methods needs to be locked:

```
public void OnDataReceived(double data)
{
	lock(m_lock)
	{
		buffer.Add(data);
	}
}
```

We also need to add it in the OuputData method

That way we will make sure we will never write at the same time we are reading into the array.

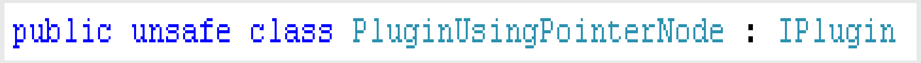
Advanced Programming

Pointers

A pointer is a special datatype, which references an value using it's address in memory.

As c# uses it's own memory space, support for pointers is quite limited.

To be able to use a pointer in c#, we need to notify that we gonna use unsafe code:

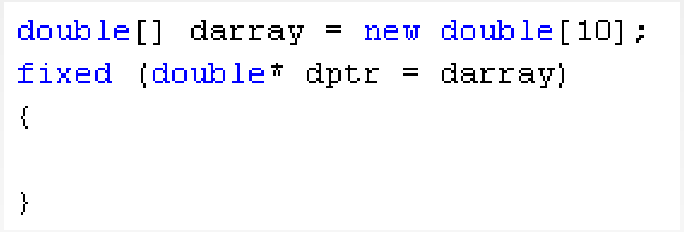


```
public unsafe class PluginUsingPointerNode : IPlugin
```

Also the program needs to be compiled with /unsafe option (Available in project properties).

Due to direct memory access, pointers can offer large performance optimizations, at the expense of losing .NET garbage collector, and we need to make sure we manage the memory ourselves.

Then to create a pointer locating to variable d (double), we can use this code:



```
double[] darray = new double[10];
fixed (double* dptr = darray)
{

}
```

The construct ”fixed” makes sure .NET will not move darray in memory while we fixed it. The variable dptr now points to darray[0].

Pointers in VVVV

Pins data in VVVV can be accessed in two ways:

- COM way: using the standard IValueIn.Getslice
- Pointer way: using IValueIn.GetValuePointer

Using pointer is useful when we have large spreadcounts, as it avoids to use multiple COM calls and we can read/write data on the fly.

We need to be careful of not trying to access and element outside of the array (Which can return wrong value, or create an Access Violation → Crash)

It's especially useful when we know the output Spread Count in advance (as we can't easily rebuild a pointer). If we need to build dynamic lists, Marshal will come to help.

Advanced Programming

Optimization Tricks

There is a lot of different way to optimize node speed. While most of the time they are not too much needed, they can prove handy in some cases.

Mod operator:

A getslice in vvvv is implemented that way: return value = index % slicecount. That way we make sure we never go out of bounds. When we use pointers we have to manage that on our own.

That can provide some interesting optimization techniques. They are generally efficient for a certain amount of spreadcount:

- float/int/double conversions: all of those are expensive too, so keeping one int variable will save some time.
- Multiply against divide : while a/b is mathematically equivalent to a*(1/b), this is not true progamming wise, so saveing divisions is always a good thing.
- Vectorization: Very popular in low level c++ programming (with MMX/SSE programming). For some operations like normalize a vector we can reduce the number of mod operations to 3 whatever the spreadcount.
- Associative operator index switching: bit more vvvv specific, but we use the mathematical concept that some math operations are associative (eg: a+b = b+a) in that case using a simple index sorting we can reduce the number of mods by 1/spreadcount where n is the input count (which for high spread count and some simple operations can increase performances significantly)
- Structures array unboxing: not a big fan of that one but this can offer some really great optimizations in some cases, but at the expense of losing a lot of readability. Instead of generating single unit data structures, we can use array a structures members, which can save a for loop. Examples of usage are animation filters/particles.

Custom Datatypes

Since beta22 vvvv allows to manage your own datatypes. Previous releases required a lot of implementation details, since 24 process has been much streamlined.

A custom datatype can simply be set up as Ispread<T>.

So let's remember our little Shapes as the beggining of the workshop.

To create a spread of Box, we can simply have the line:

```
[Output("Output")]
private ISpread<Box> FpinOutput;
```

Similarly, to create a spread of Circles:

```
[Output("Output")]
private ISpread<Circle> FpinOutput;
```

Now we can also have an Input which will accept both Boxes and Circles:

```
[Input("Input")]
private ISpread<IShape2dHitTester> FpinInput;
```

This works because every shape does implement IShape2dHitTester

If we did:

```
[Input("Input")]
private ISpread<Circle> FpinInput;
```

Then only shapes of type Circle would be accepted.

Advanced Programming

Custom Datatypes

Some guidelines using Custom Types:

- Try to always output the class at the bottom of the hierarchy. It's better to output Box than Ishape2dHitTester, as vvvv will recognize that Box implements the interface. This allows to create custom nodes that operates only on Boxes, and other on any type of shapes, all in a type safe way.
- Remember that classes are just references, so modifying the class internal state in a bottom node will update in the top one. This can lead to discrepancies if more than one output is connected.
- Be careful of objects lifetime. (For example passing a running thread is generally not a good idea, better to send events). It is also true when your classes wraps unmanaged resources.

Exercise: Generic Hit Tester

The main concept is to generate Shape objects (one node per shape type):

- Box (2d.Shape)
- Circle (2d.Shape)

Then we will implement a hit test against these Shapes (This nodes should accept any shape type):

- HitTest (2d.Shape) (Will perform Hits Slicewise)

Additional Exercises:

- Use a Generics based abstract class for Shape generation
- Create a Hittest version which compares every point to every shape.
- Create a Cons (2d.Shape). This will output the Base shape definition (as it can mix types).
- Create some other shapes of your choice (Triangle,Polygon,Superformula...)