# .NET, Visual Studio, C#, and Kinect SDK Installation

Anant Bhardwaj

# Goals

- Introduction to .NET and Microsoft Visual Studio
- Introduction to C#
- Provide enough information to allow you to follow a C# code.
- Intro to WPF and XAML
- Setting-up Kinect SDK
- Running Kinect SDK Samples
- Kinect SDK Code Walkthrough

# Introduction to .NET

- .NET is:
  - Microsoft's Platform for Windows Development
  - CLR (Common Language Runtime) – the Virtual Machine that runs MSIL (Microsoft Intermediate Language Code)
  - A set of compilers that can generate MSIL
    - C#, Visual Basic, C++ etc..
  - This provides inter-operability

# Microsoft Visual Studio

- The full development system for Microsoft platforms (Windows, Windows Mobile, Silverlight etc…)
  - provides a nice editor window with auto-complete, intellisense etc…
  - integrates compiler, linker, debugger etc…
  - optimizations, data access, multi-language etc…

# Different Types of Applications

- Console Application
  - has standard streams (out, in, err)
  - GUI can be added manually
- Windows Application
  - GUI based
  - no standard streams (out, in, err)
  - the main thread is shared by the GUI message pump & your code
- Service
  - no standard streams (out, in, err)
  - main thread is commandeered by the windows service manager
  - no GUI (runs in session 0 – service session)

# Introduction to C#

- A simple, modern, general-purpose object-oriented language
- Originated by Microsoft as a response to Java
- Software robustness and programmer productivity
  - strong type checking, array bounds checking, detection of use of uninitialized variables, source code portability, automatic garbage collection
  - allows rapid application development
- Developed specifically for .NET
- Very easy to migrate from C++ and Java
- We assume that you know Java and thus we will ignore the things that are similar to Java.

# C# Hello CS247L

```
using System;
namespace hci{
    public class HelloCS247L {
        public static void Main(string[] args) {
            Console.Write ("Hello  ");
            if(args.Length > 0 ){
                Console.Write (“{0}", args[0]);
                for (int i=1; i < args.Length; i++) {
                    Console.Write(“, {0}", args[i]);
                }
            }
            Console.WriteLine (“Welcome to CS247L!");
        }
    }
}
```

# Class Declaration and Namespace

- The **namespace** keyword is used to declare a scope. This namespace scope lets you organize code and gives you a way to create globally unique types.

```
using System;
namespace stanford.cs {

        namespace hci {
                class CS247{…..}
        }
        namespace ai {
                class CS221{…..}
        }

        class ComputerForum{……}

 }
```

# Built-in data types

- Integer Types
  - sbyte, byte,  short, ushort, int, uint, long, ulong
- Real Numbers
  - float, double, decimal
- Text
  - char, string
- Boolean
  - bool
- Object
  - object

# C# parameters passing

- In C#, method parameters that refer to an object are always passed by reference, while primitive data type parameters are passed by value.

- To pass primitive data type by reference, you need to specify one of the keywords **ref** or **out**.

- A **ref** parameter must be initialized before use, while an **out** parameter does not have to be explicitly initialized before being passed and any previous value is ignored.

# Arrays

- Single-dimensional arrays

```
int[] numbers = new int[100];
int[] numbers = {1, 2, 3, 4, 5};
```

- Multidimensional array with fixed dimension's sizes.

```
int[,] arr = new int[2,3];
int[,] arr = {{1, 2, 3}, {4, 5, 6}};
```

- Multidimensional array with irregular dimensions' sizes.

```
int[][] arr = new int[2][];
arr[0] = new int [4];
arr[1] = new int [6];
int[][] arr = new int[][] {new int[] {1, 2, 3, 4}, new int[] {5, 6, 7, 8, 9, 10}};
```

# Properties

```
using System;
class CS247 {
    private int num_students = 0;
    public string Num_Students {
    get {
        return num_students;
    }
    set {
        num_students = value;
    }
}
```

# Indexers

- Indexers is used to provide array-like access to the classes.
- Let's open IndexerAndProperties project

```
using System;

class CS247 {
   private string[] students;

   public CS247(int size) {
      students = new string[size];

      for (int i=0; i < size; i++)  {
         students[i] = "Student #"+i;
      }
   }
  //contd…..
```

# Indexers

```
public string this[int pos] {
        get  {
            return students[pos];
        }
        set {
            students[pos] = value;
        }
   }

 static void Main(string[] args)  {
        int size = 10;
       CS247 cs247 = new CS247 (size);
        cs247[0] = “Arti”;
        cs247[1] = “Daniel”;
        for (int i=0; i < size; i++){
            Console.WriteLine(“cs247[{0}]: {1}”, i, cs247[i]);
        }
   }
}
```

# Delegates and Events

- **Delegates** are reference types which allow indirect calls to methods.
  - A **delegate instance** holds **references to some number of methods**, and by invoking the delegate one causes all of these methods to be called.
  - The usefulness of delegates lies in the fact that the functions which invoke them are blind to the underlying methods.
- It can be seen that delegates are functionally rather similar to C++'s **function pointers**. However, it is important to bear in mind two main differences.
  - Firstly, **delegates are reference types** rather than value types.
  - Secondly, some **single delegates can reference multiple methods**.

# Delegates and Events

- Delegate declaration

  public delegate void Print (String s);

- Suppose, for instance, that a class contains the following method:

```
public void realMethod (String myString) {
   // method code
}
```

- Another method in this class could then instantiate the Print delegate in the following way, so that it holds a reference to realMethod;

  Print delegateVariable = new Print(realMethod);

# Delegates and Events

- We can note two important points about this example. Firstly, the unqualified method passed to the delegate constructor is implicitly recognized as a method of the instance passing it. That is, the code is equivalent to:

  ```
  Print delegateVariable = new Print(this.realMethod);
  ```

- We can, however, in the same way pass to the delegate constructor the methods of other class instances, or even static class methods. In the case of the former, the instance must exist at the time the method reference is passed. In the case of the latter (exemplified below), the class need never be instantiated.

  ```
  Print delegateVariable = new Print (ExampleClass.exampleMethod);
  ```

# Delegates and Events

- In C#, **‘Event’** is a class member that is activated whenever the event it was designed for occurs. Anyone interested in the *event* can register and be notified as soon as the *event* fires. At the time an *event* fires, registered methods will be invoked.
- ***Events*** and ***delegates*** work hand-in-hand to provide a program's functionality. It starts with a class that declares an *event*.
- Any class, including the same class that the *event* is declared in, may register one of its methods for the *event*. This occurs through a *delegate*, which specifies the signature of the method that is registered for the *event*.
- The *delegate* may be one of the pre-defined .NET *delegates* or one you declare yourself. Whichever is appropriate, you assign the *delegate* to the *event*, which effectively registers the method that will be called when the *event* fires.

# Delegates and Events

- Let’s open DelegatesAndEvents project

# Let’s do something fun

- A TickEvent class that will generate an event every second
- Two different clocks that implement different time-zones. They should use the events from TickEvent to advance the clock.
- Use
  - properties to set and get the time,
  - indexer for different clocks,
  - delegates and events for advancing the clock

# Useful Namespaces to know

- System
  - namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
- System.Collections.Generic
  - contains interfaces and classes that define generic collections (list, dictionary etc..), which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.
- System.IO
  - contain types that support input and output, including the ability to read and write data to streams, to compress data in streams, to create and use isolated stores, to map files to an application's logical address space, to store multiple data objects in a single container, to communicate using anonymous or named pipes, to implement custom logging, and to handle the flow of data to and from serial ports.

# Useful Namespaces to know

- System.Windows
  - namespaces contain types used in Windows Presentation Foundation (WPF) applications, including animation clients, user interface controls, data binding, and type conversion.
  - System.Windows.Forms and its child namespaces are used for developing Windows Forms applications.
- System.Threading
  - contain types that enable multithreaded programming. A child namespace provides types that simplify the work of writing concurrent and asynchronous code.
- System.Data
  - contain classes for accessing and managing data from diverse sources. The top-level namespace and a number of the child namespaces together form the ADO.NET architecture and ADO.NET data providers. For example, providers are available for SQL Server, Oracle, ODBC, and OleDB.

# Windows Presentation Foundation (WPF) applications

- **Windows Presentation Foundation** (or **WPF**) is a graphical subsystem for rendering user interfaces in Windows-based applications

- WPF attempts to provide a consistent programming model for building applications and provides a separation between the user interface and the inner logic.

- WPF employs XAML, a derivative of XML, to define and link various UI elements.

- WPF utilizes DirectX.

# XAML

- is a declarative markup language. As applied to the .NET Framework programming model, XAML simplifies creating a UI for a .NET Framework application.
  - XAML code is short and clear to read
  - Separation of designer code and logic
  - Graphical design tools like Expression Blend require XAML as source.
  - The separation of XAML and UI logic allows it to clearly separate the roles of designer and developer.

# C# vs. XAML

```
StackPanel stackPanel = new StackPanel();
this.Content = stackPanel;
Button Button button = new Button();
button.Margin= new Thickness(20);
button.Content = "OK";
stackPanel.Children.Add(button);
```

==========

```
<StackPanel>
<Button Margin="10" HorizontalAlignment="Right">OK</Button>
</StackPanel>
```

==========

# Creating a UI in XAML

- At the beginning of every XAML file you need to include two namespaces.

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="hci.CS247">
 <Button Click="onClick" >Click Me!</Button>
</Page>

namespace hci {
        public partial class CS247 {
                void onClick(object sender, RoutedEventArgs e)
                {
                        Button b = e.Source as Button;
                         b.Foreground = Brushes.Red;
                }
        }
}
```

# In-lining the code

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class=“hci.CS247" >
<Button Name="button1" Click=“onClick">Click Me!</Button> <x:Code>
<![CDATA[
void onClick(object sender, RoutedEventArgs e)
{
        button1.Content = "Hello World";
}
]]>
</x:Code>
</Page>
```

# XAML Styling

```
<Window.Resources>
...

<!--A Style that affects all TextBlocks-->
<Style TargetType="TextBlock">
<Setter Property="HorizontalAlignment" Value="Center" />
<Setter Property="FontFamily" Value="Comic Sans MS"/>
<Setter Property="FontSize" Value="14"/>
</Style>

...
</Window.Resources>
```

# Setting up Knect SDK

- **Download Kinect for Windows SDK**
  - http://www.kinectforwindows.org
- Download **DirectX**
  - Microsoft DirectX® SDK - June 2010 or later version
  - Current runtime for Microsoft DirectX® 9
- **Download Microsoft Speech SDK**
  - Microsoft Speech Platform Runtime, version 10.2 – select 32-bit if you are running 32-bit Windows. If you have 64-bit Windows, we you need to install both the 32-bit and 64-bit runtime.
  - Microsoft Speech Platform - Software Development Kit, version 10.2 – select 32-bit or 64-bit according to your Windows installation
  - Kinect for Windows Runtime Language Pack, version 0.9

# Testing the installation

- Run the kinect sample binaries located at
  - C:\Program Files\Microsoft SDKs\Kinect\v1.0 Beta2\Samples
    - \bin
      - KinectAudioDemo.exe
      - ShapeGame.exe
      - Skeletal-Viewer-WPF.exe

# Let’s compile/run the sample code

- Open the solution file
- C:\Program Files\Microsoft SDKs\Kinect\v1.0 Beta2\Samples\Managed\ KinectSDKSamples(C#).sln

# Samples -- Basic Code Walkthrough

# Where to go for more details

- C#:
  - http://msdn.microsoft.com/en-us/library/aa288436(v=VS.71).aspx
- WPF Fundamentals:
  - http://msdn.microsoft.com/en-us/library/ms746927.aspx
- XAML Overview:
  - http://msdn.microsoft.com/en-us/library/ms752059.aspx
- Setting up Kinect SDK
  - http://channel9.msdn.com/Series/KinectSDKQuickstarts/Getting-Started

Next Lab

# Kinect SDK in Detail