Kinect Server C# Walkthrough:

Abstract The FRC Kinect Server uses the natural user interface (NUI) API in the Kinect™ for Windows® Software Development Kit (SDK) Beta to capture, process, send and optionally render depth, video, and skeleton data for use in the FIRST Robotics Competition.

Resources For a complete list of documentation for the Kinect for Windows SDK Beta, plus related reference and links to the online forums, see the beta SDK website at:
http://kinectforwindows.org

In this guide

v2.0 – December 19, 2011

Contents

PART 1—Introduction

The Kinect™ for Xbox 360® sensor includes cameras that deliver depth information, color data, and skeleton tracking data. The natural user interface (NUI) API in the Kinect for Windows® Software Development Kit (SDK) Beta enables applications to access and manipulate this data.

The Kinect Server captures the depth stream, color stream, and skeletal tracking frames, processes the skeletal stream into packets sent to the FRC Driver Station application and optionally displays the streams on screen.

By default the Kinect Server hides its window for use with the Driver Station and Dashboard applications.

When you run the Kinect Server in debug mode (-debug flag), you can see the following:

- The depth stream, which shows background in gray scale and different people in different colors. Darker colors indicate objects that are farther from the camera.
- Tracked skeletons of up to two people who have been detected within the frame.
- The color video stream, which shows the red-green-blue (RGB) image from the Kinect sensor. (if also using the –color flag)
- The rate at which captured frames are delivered to the application (logged to a file if using the –logFPS flag)

If moving figures are too close to the camera, unreliable or odd images might appear in the skeleton and depth views. The optimal range is 2.6 to 13.12 feet (0.8 to 4 meters). The depth and skeleton views detect people only if the entire body fits within the captured frame.

Getting Started with Modifying Gestures

This Walkthrough is designed to describe and explain all of the major components of the Kinect Server. It is anticipated that many teams will want to modify or create new gestures without making any changes to the other parts of the Server. The **FIRSTGestureProcessor.cs** section of this document will walk you through the code for the FIRST default gestures, then explain how to begin modifying or creating your own gestures.

Opening and Compiling the Kinect Server

To open the Kinect Server code in Visual Studio:

1. Locate the **Source** folder in **Program Files\FRC Kinect Server** (in **Program Files (x86)** on 64-bit versions).
2. It is recommended to copy this folder to a new location before proceeding; this folder will be overwritten if any updates to the Kinect Server are released.
3. Inside the **Source** folder, open the **Kinect Server** folder and locate the **Kinect Server.sln** file. Double click that file to open the project in Visual C# Express.

To build and run the Kinect Server:

1. Select **Build**->**Build Solution** OR press F6 to build your application
2. Press **CTRL+F5** to run the built application.
3. To use your new Kinect Server with the Driver Station
    a. Browse to **Program Files\FRC Kinect Server** (For 64-bit machines use **Program Files(x86)**) and rename the **KinectServer.exe** and **Microsoft.Samples.Kinect.WpfViewers.FIRST.dll** files to keep them as backups.
    b. Copy **KinectServer.exe** AND **Microsoft.Samples.Kinect.WpfViewers.FIRST.dll** from the **Kinect Server\bin\Release** folder where your source code is located to **Program Files\FRC Kinect Server** (For 64-bit machines use **Program Files(x86)**)

PART 2—Kinect Server Code Walkthrough

The C# Kinect Server uses the managed NUI API to capture depth data, color, and skeletal tracking data, processes the skeleton data into joystick data using a set of default gestures and sends the resulting packets, and optionally renders the images on the screen by using WPF. For an introduction to WPF, see "Resources" in Part 4 of this document.

Program Basics

The Kinect Server source code is installed in the FRC Kinect Server folder in the Program Files directory (Program Files (x86) on 64-bit Windows). The Kinect Server application is made up of quite a few files, this document will cover only the major ones:

- App.xaml declares application-level resources.
- App.xaml.cs contains the code behind app.xaml. This file contains a startup function that is used to set different modes for the Kinect Server.
- MainWindow.xaml declares the layout of the main application window.
- MainWindow.xaml.cs contains the code behind the main window, which implements Kinect initialization and management, networking initialization, and display functions
- Version0Manager.cs manages the sending of the packets from the Kinect Server to the Driver Station
- KinectUtils.cs contains some helper methods used with Kinect data
- FIRSTGestureProcessor.cs implements the processing of the skeleton into Joystick data using a default set of gestures.

To build and run the Kinect Server

1. In Windows Explorer, navigate to the FRC Kinect Server\Source\Kinect Server directory.
2. Double-click the icon for the .sln (solution) file to open the file in Visual Studio.
3. Build the application.
4. Press CTRL+F5 to run the sample.

The solution file for the sample targets the x86 platform, in order to distribute a single executable for all teams

App.xaml and App.xaml.cs

The App.xaml file declares application level resources. The version of this file used in the Kinect Server has been modified slightly from the typical WPF App.xaml file.

Default App.xaml line:

```
StartupUri="MainWindow.xaml"
```

Kinect Server App.xaml line:

```
Startup="Application_Startup"
```

This change runs a method in App.xaml.cs when the program is launched instead of directly launching the MainWindow. This is necessary to hide the MainWindow when running in the default mode for use with the Driver Station and to process command line arguments to activate certain features.

The *Application_Startup* method is contained in App.xaml.cs:

```
    private void Application_Startup(object sender, StartupEventArgs e)
    {
        var mainWindow = new Edu.FIRST.WPI.Kinect.KinectServer.MainWindow();

        mainWindow.ShowInTaskbar = false;
        mainWindow.ShowActivated = false;

        foreach (String s in e.Args)
        {
            switch (s)
            {
                case "-debug":
                    mainWindow.minimal = false;
                    mainWindow.ShowInTaskbar = true;
                    mainWindow.ShowActivated = true;
                    break;
                case "-logFPS":
                    mainWindow.logFPS = true;
                    break;
                case "-color":
                    mainWindow.renderColor = true;
                    break;
                default:
                    break;
            }
        }
        mainWindow.Show();
        if(mainWindow.minimal)
            MainWindow.Visibility = Visibility.Hidden;
    }
```

The *Application_Startup* method sets up parameters based on the desired mode and then launches the MainWindow:

- **ShowInTaskbar** and **ShowActivated** are parameters of the MainWindow that determine whether it is shown in the Taskbar and whether it takes focus when it is initially shown.
- **e.Args** contains the arguments passed to the application when it is launched. If any arguments have been passed to the application, this method runs through each of them and compares them to the 3 valid arguments.
- **minimal** is a Boolean defined in the MainWindow.xaml.cs file. It is set to true when running in the hidden Driver Station mode. Its use will be discussed further in the walkthrough of that file.
- **logFPS** is a Boolean defined in the MainWindow.xaml.cs file. It indicates whether to log the skeleton framerate to a file. By default this logging is turned off.
- **renderColor** is a Boolean defined in the MainWindow.xaml.cs file. It indicates whether to show the color video stream or not. By default the display of the color stream is turned off.
- Regardless of mode, the MainWindow must be shown in order to launch the code contained in MainWindow.xaml.cs and also to create a Window Handle which is used by the Kinect SDK code.
- After showing the MainWindow, if the application is not in debugging mode, the window is set to hidden.

MainWindow.xaml

The *KinectServer* namespace declares a public **Window** class object that is named *MainWindow*, which contains the following:

- Code to initialize global variables.
- The *Window_Loaded* method, which handles the **Window.Loaded** event, sets the process priority, kills any other instances of the Kinect Server and initializes the Kinect and packet sender
- The *Window_Closed* method, which handles the **Window.Closed** event and cleans up the window, event handlers, logfile (if used), and Kinect.
- The *ShowStatus* method which dictates how the Kinect status is displayed
- The methods for management of Kinect devices *InitializeKinectServices* and *UninitializeKinectServices, KinectStart* and *KinectStop, IsKinectStarted, KinectDiscovery,* and the *KinectStatusChanged* event handler.
- The *IsKinectStarted*, and *IsSkeletalViewerAvailable* properties
- The *SkeletonsReady* event handler, called whenever a skeleton frame is received
- The *CalculateFrameRate* method used when logging the frame rate
- The *UpdateUIBasedOnKinectCount* method
- The KinectViewer methods *AddKinectViewer, RemoveKinectViewer, DisableOrAddKinectViewer, CreateAllKinectViewers, CleanUpAllKinectViewers, and FindViewer*

Members and Constructor

The *MainWindow* class initializes the window and then creates and initializes global variables as follows:

- Declares the Booleans *minimal, logFPS, and renderColor*. See the **App.xaml** section above for descriptions of these flags.
- Declares *Kinect* as a public **Runtime** object, which represents the Kinect sensor instance.
- Defines an enum *ErrorCondition* to easily pass Kinect status information around
- Declares *runtimeOptions* as a **RuntimeOptions** object to hold the desired options to use with the Kinect
- Declares *manager* as a **Version0Manager** object, which handles packet creation and sending
- Declares *otherServers* as a vector of **Process** objects to hold all instances of the KinectServer process that are found.
- Declares a private **Runtime** object, _Kinect which represents the Kinect sensor instance. The purpose for the public and private **Runtime** objects is explained below where the *Kinect* get and set accessors are covered.
- Initializes several variables — *lastTime* , *framerate*, *lastFrames*, and *totalFrames* — that are used to calculate the number of frames per second.
- Declares *fpsLog* as a **StreamWriter** object which is used to log the framerate

The following shows the initialization code from *MainWindow*:

```
namespace Edu.FIRST.WPI.Kinect.KinectServer
{
   public partial class MainWindow : Window
   {

      public MainWindow()
      {
         InitializeComponent();
      }

      public bool minimal = true;
      public bool logFPS = false;
      public bool renderColor = false;

      public Runtime Kinect…

      public enum ErrorCondition…

      #region Private state
      RuntimeOptions runtimeOptions;
      Version0Manager manager;
      Process[] otherServers;
      Runtime _Kinect;
      DateTime lastTime = DateTime.MaxValue;
      int frameRate = 0;
      int lastFrames = 0;
      int totalFrames = 0;
      StreamWriter fpsLog;

      #endregion Private state
```

Kinect object

The public **Runtime** object *Kinect* has customized *get* and *set* accessors. These make *Kinect* an alias to the private *_Kinect*, while ensuring that the Kinect device is always initialized and uninitialized when appropriate:

```
      public Runtime Kinect
      {
         get
         {
            return _Kinect;
         }
         set
         {
            if (_Kinect != null)
            {
               UninitializeKinectServices(_Kinect);
            }
            _Kinect = value;
```

```
            if (_Kinect != null)
            {
                InitializeKinectServices(_Kinect);
            }
        }
    }
```

ErrorCondition enum

The enum **ErrorCondition** is used to simplify the passing of the Kinect status from one part of the code to another

```
    public enum ErrorCondition
    {
        None,
        NoPower,
        NoKinect,
        NotReady,
        KinectAppConflict,
        EngConflict,
    }
```

The use and meaning of the various statuses is described below in the parts of the code where they appear.

Window Events

Window_Loaded

The *Window_Loaded* method sets the process priority, kills any other instances, optionally opens the log file, grabs the version of the SDK DLL, initializes the **manager** object to handle packet creation and sending, starts the Kinect, and optionally creates the KinectViewers as follows:

```
private void Window_Loaded(object sender, EventArgs e)
{
    Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.BelowNormal;

    otherServers = System.Diagnostics.Process.GetProcessesByName("KinectServer");
    foreach (System.Diagnostics.Process killThis in otherServers)
    {
        if (killThis.Id != Process.GetCurrentProcess().Id)
        {
            killThis.Kill();
        }
        System.Threading.Thread.Sleep(500); //wait to allow any instance using a Kinect to release it
    }

    if (logFPS)
    {
        fpsLog = new
            StreamWriter(Environment.GetFolderPath(Environment.SpecialFolder.CommonDocuments) +
            "\\FRC\\KinectFPS.txt", false);
    }

    manager = new Version0Manager("01.07.12.00", "localhost", 1155);
    KinectStart();

   if (!minimal)
    {
        AddKinectViewer(Kinect);
        UpdateUIBasedOnKinectCount();
    }
}
```

- The process is set to a priority of BelowNormal in order to prevent the KinectServer from hogging the CPU on processor limited machines and affecting the operation of the Driver Station application.
- The Kinect Server then searches for all instances of itself currently running, and kills any other instances. Only one application can use the Kinect at a time, so only a single instance should be allowed to run at once.
- If framerate logging is enabled by passing the –logFPS argument, the Server opens the log file, overwriting any existing file.
- The Server initializes the *manager* object. The constructor used here for **Version0Manager** takes 3 parameters, the string to use as the version (in this case a formatted version of the Kickoff date), the destination IP for the data, and the destination port for the data. The KinectServer uses port 1155 on the localhost to send the data to the Driver Station. This code can be changed to the alternate constructor to use your own Gesture Processor that implements the IGestureProcessor interface.

- The **KinectStart** method is called to detect and initialize the Kinect. This also sets up appropriate event handlers for Kinect status changes which including plugging in a Kinect if one is not initially present.
- Finally, if the Kinect Server is running in debugging mode the KinectViewers are created and the UI is updated.

The Kinect Server implements the event-driven model of receiving frames from the Kinect. After this initialization code is complete, all other processing done by the Kinect Server is triggered by various events.

Window_Closed

The *Window_Closed* event handler is called when the window is closed. This can be triggered in a variety of ways such as a signal from another application, a signal from Windows, or by pressing the red 'X' in the corner of the window:

```
private void Window_Closed(object sender, EventArgs e)
{
    CleanUpAllKinectViewers();
    if (logFPS)
        fpsLog.Close();

    KinectStop();
}
```

- All remaining Kinect viewers are cleaned up and removed from the UI
- If open, the log file is closed
- The *KinectStop* method is called, removing the status event handlers and closing the Kinect if one is open.

Kinect Management

The Kinect Server has a number of different functions used to manage the Kinect device(s)

ShowStatus

The first such method is the *ShowStatus* method:

```
private void ShowStatus(ErrorCondition errorCondition)
{
   manager.SetKinectStatus(errorCondition);
}
```

This method takes the status and uses it to set the Kinect status of the *manager* object. This method may seem pointless, but the purpose is to allow for extension of the status notification methods (adding display on a GUI, logging to a file, etc.) without changing any of the Kinect handling code that calls this method.

InitializeKinectServices

The *InitializeKinectServices* method initializes the Kinect device for use

```
private void InitializeKinectServices(Runtime runtime)
{
   bool skeletalViewerAvailable = IsSkeletalViewerAvailable;

   runtimeOptions = skeletalViewerAvailable ?
                    RuntimeOptions.UseDepthAndPlayerIndex | RuntimeOptions.UseSkeletalTracking
                    RuntimeOptions.UseColor
                    : RuntimeOptions.UseDepth | RuntimeOptions.UseColor;

   try
   {
      runtime.Initialize(runtimeOptions);
   }
   catch (COMException comException)
   {
      if (comException.ErrorCode == -2147220947)  //Runtime is being used by another app.
      {
         runtime = null;
         ShowStatus(ErrorCondition.KinectAppConflict);
         return;
      }
      else
      {
         throw comException;
      }
   }

   if (runtimeOptions.HasFlag(RuntimeOptions.UseSkeletalTracking))
   {
      Kinect.SkeletonEngine.TransformSmooth = true;
      Kinect.SkeletonFrameReady += new
         EventHandler<SkeletonFrameReadyEventArgs>(SkeletonsReady);
   }
}
```

- Only a single instance of the Skeletal Engine can be running at a time so the code checks if the engine is in use and sets the *runtimeOptions* to not use skeletal tracking if it is in use.
- Only one application can access a particular device at a time so if the attempt to initialize the Kinect fails with a particular error code, the server knows that another application is using the Kinect and sets the status accordingly.
- If the Skeletal Tracking is enabled, the smoothing is turned on and the event handler is registered for the *SkeletonFrameReady* event.

IsSkeletalViewerAvailable

The *isSkeletalViewerAvailable* property is defined further down in the code and simply loops through the detected Kinect devices and checks if any of them have assigned the SkeletonEngine:

```
private bool IsSkeletalViewerAvailable
{
   get { return Runtime.Kinects.All(k => k.SkeletonEngine == null); }
}
```

UninitializeKinectServices

The *UnitializeKinectServices* method is much simpler than initializing:

```
private void UninitializeKinectServices(Runtime runtime)
{
   runtime.Uninitialize();
   runtime.SkeletonFrameReady -= new
        EventHandler<SkeletonFrameReadyEventArgs>(SkeletonsReady);
}
```

KinectStart

The *KinectStart* method is designed to be called once at the start of the application:

```
private void KinectStart()
{
   KinectDiscovery();

   //listen to any status change for Kinects
   Runtime.Kinects.StatusChanged += new
         EventHandler<StatusChangedEventArgs>(Kinects_StatusChanged);

   if (Kinect == null)
   {
      if (Runtime.Kinects.Count == 0)
      {
         ShowStatus(ErrorCondition.NoKinect);
      }
      else
      {
         if (Runtime.Kinects[0].Status == KinectStatus.NotPowered)
         {
            ShowStatus(ErrorCondition.NoPower);
         }
      }
   }
```

```
        else if (!runtimeOptions.HasFlag(RuntimeOptions.UseSkeletalTracking))
            ShowStatus(ErrorCondition.EngConflict);
        else
            ShowStatus(ErrorCondition.None);
    }
```

- The *KinectDiscovery* method is called to search through any connected Kinect devices and initialize the first one found with no error
- An event handler is registered for the *StatusChanged* event which will be called whenever a Kinect is plugged in, unplugged, has the power disconnected, or has the power connected.
- If no Kinect was set by *KinectDiscovery*, the appropriate error status is set
- If a Kinect was initialized without the SkeletalTracking option, an error status is set indicating that the Skeletal Engine is in use by another application.

KinectStop

KinectStop is simpler than *KinectStart*, simply removing the StatusChanged event handler and setting the Kinect to null if it is not already:

```
    private void KinectStop()
    {
        Runtime.Kinects.StatusChanged -= new
            EventHandler<StatusChangedEventArgs>(Kinects_StatusChanged);
        if (Kinect != null)
        {
            Kinect = null;
        }
    }
```

KinectDiscovery

The *KinectDiscovery* method loops through any attached Kinect devices and initializes the first one found without error:

```
    private void KinectDiscovery()
    {
        foreach (Runtime kinect in Runtime.Kinects)
        {
            if (kinect.Status == KinectStatus.Connected)
            {
                if (Kinect == null)
                {
                    Kinect = kinect;
                    return;
                }
            }
        }
    }
```

Kinects_StatusChanged

The *Kinects_StatusChanged* event handler is called whenever a Kinect is plugged or unplugged, or has power plugged or unplugged.

```
Private void Kinects_StatusChanged(object sender, StatusChangedEventArgs e)
{
    switch (e.Status)
    {
        case KinectStatus.Connected:
            if (Kinect == null)
            {
                Kinect = e.KinectRuntime;
                if (Kinect == null)
                    ShowStatus(ErrorCondition.KinectAppConflict);
                else
                    ShowStatus(ErrorCondition.None);
                CleanUpAllKinectViewers();
                if (!minimal)
                    AddKinectViewer(Kinect);
            }
            break;
        case KinectStatus.Disconnected:
            if (Kinect == e.KinectRuntime)
            {
                Kinect = null;
            }
            ShowStatus(ErrorCondition.NoKinect);
            RemoveKinectViewer(e.KinectRuntime);
            break;
        case KinectStatus.NotReady:
            ShowStatus(ErrorCondition.NotReady);
            break;
        case KinectStatus.NotPowered:
            if (Kinect == e.KinectRuntime)
            {
                Kinect = null;
            }
            ShowStatus(ErrorCondition.NoPower);
            if (!minimal)
                DisableOrAddKinectViewer(e.KinectRuntime);
            break;
        default:
            throw new Exception("Unhandled Status: " + e.Status);
    }
    UpdateUIBasedOnKinectCount();
}
```

- If a Kinect has been connected, and there is not a Kinect currently in use, the code attempts to initialize the connected device. It sets the status based on whether that operation succeeds, and then resets the KinectViewers.
- If a Kinect has been disconnected, the Kinect is uninitialized, the viewer is removed, and the status is set to No Kinect.
- If a Kinect is indicated as "NotReady" the status is set to match
- If a Kinect is indicated to have no power, it is uninitialized, the status is set, and a Viewer is either disabled or added depending on whether a Viewer exists for that Kinect or not.
- The UI is then updated based on the number of connected devices

IsKinectStarted

The *IsKinectStarted* property is used to indicate if there is a currently initialized Kinect:

```
private bool IsKinectStarted
{
    get { return Kinect != null; }
}
```

Process Skeleton Data

SkeletonsReady

When a frame of Skeleton Data is ready the Kinect SDK signals the **SkeletonFrameReady** event, in the Kinect Server this event is handled by the *SkeletonsReady* method (and optionally by the *nui_SkeletonFrameReady* method in the KinectDiagnosticViewer):

```
void SkeletonsReady(object sender, SkeletonFrameReadyEventArgs e)
    {
        SkeletonFrame skeletonFrame = e.SkeletonFrame;

        if (skeletonFrame == null)
        {
            return;
        }

        if (logFPS)
            CalculateFrameRate();
        manager.ProcessSkeleton(skeletonFrame);
    }
```

- If the SkeletonFrame passed in to the method is null, the function returns without doing any processing, this can occur when a Kinect is disconnected with the current SDK.
- If the SkeletonFrame is valid, the code then calculates the framerate if FPS logging is enabled.
- Finally the code calls the *ProcessSkeleton* method of the *manager* object which processes the skeleton into the data sent out to the DriverStation.

CalculateFrameRate

The *CalculateFrameRate* method is used to calculate the frame rate of the Skeleton data if FPS logging is enabled:

```
private void CalculateFrameRate()
    {
        ++totalFrames;

        DateTime cur = DateTime.Now;
        if (lastTime == DateTime.MaxValue || cur.Subtract(lastTime) > TimeSpan.FromSeconds(1))
        {
            frameRate = totalFrames - lastFrames;
            lastFrames = totalFrames;
            lastTime = cur;
            fpsLog.WriteLine(frameRate);
            fpsLog.Flush();
        }
    }
```

- On most calls this method will simply increment the total number of frames
- On the first call or if >1 second has passed a framerate will be calculated by taking the current number of frames and subtracting the previous number
- The lastFrames variable is then updated to the current total and the lastTime variable is updated to the current time
- The frame rate is then written to the log file and Flush() is called to force the write to happen immediately, this is done so a valid log file is written even if the Server closes unexpectedly

UI & Kinect Viewers

When in debugging mode the Kinect Server displays a GUI similar to that of the Microsoft Skeleton Viewer WPF sample program. Most of the code for that display is contained in the Microsoft.Samples.Kinect.WpfViewers.FIRST companion project, but a number of methods to manipulate the viewers are included with the MainWindow code.

UpdateUIBasedOnKinectCount

The *UpdateUIBasedonKinectCount* method is used to show or hide the message to connect a Kinect and to trigger an update to the text of the Kinect Viewer:

```
private void UpdateUIBasedOnKinectCount()
{
    switch (Runtime.Kinects.Count)
    {
        case 0:
            insertKinectSensor.Visibility = System.Windows.Visibility.Visible;
            break;
        default:
            insertKinectSensor.Visibility = System.Windows.Visibility.Collapsed;
            break;
    }
    foreach (UIElement element in viewerHolder.Items)
    {
        var kinectViewer = element as KinectDiagnosticViewer;
        if (kinectViewer != null)
        {
            kinectViewer.UpdateUi();
        }
    }
}
```

AddKinectViewer

The *AddKinectViewer* method adds a **KinectDiagnosticViewer** for the runtime passed to it:

```
private void AddKinectViewer(Runtime runtime)
{
    if(runtime != null)
    {
        var kinectViewer = new KinectDiagnosticViewer(renderColor);
        kinectViewer.RuntimeOptions = runtimeOptions;
        kinectViewer.Kinect = runtime;
        viewerHolder.Items.Add(kinectViewer);
    }
}
```

- A new KinectDiagnosticViewer is created passing renderColor as a parameter to indicate whether the viewer should display the color video stream or not
- The RuntimeOptions and Kinect of the new viewer are then set
- Now that the viewer is set up it is added to the UI as part of the vieweHolder element

RemoveKinectViewer

The *RemoveKinectViewer* method removes a KinectDiagnosticViewer for the given Kinect from the UI

```
private void RemoveKinectViewer(Runtime runtime)
{
    var foundViewer = FindViewer(runtime);

    if (foundViewer != null)
    {
        foundViewer.Kinect = null;
        viewerHolder.Items.Remove(foundViewer);
    }
}
```

- The *FindViewer* method is used to identify the Viewer associated with the given Kinect
- If a Viewer is found, the Kinect of the Viewer is set to null and the Viewer is removed from the UI

DisableOrAddKinectViewer

This method is used if the previous state of the Viewer is unknown, for example a working Kinect can have the power plugged in or a Kinect can be freshly plugged into USB without the power adapter plugged in:

```
private void DisableOrAddKinectViewer(Runtime runtime)
{
    var foundViewer = FindViewer(runtime);

    if (foundViewer != null)
    {
        runtime.Uninitialize();
    }
    else
    {
        AddKinectViewer(runtime);
    }
}
```

- If a viewer exists for the given Kinect, the Kinect was previously initialized, so it is uninitialized. The Viewer is left in place to display the status.
- If no Viewer exists for the Kinect it has just been plugged in, a Viewer is created to display the status.

CleanUpAllKinectViewers

The *CleanUpAllKinectViewers* method is used to remove all existing KinectDiagnosticViewers:

```
private void CleanUpAllKinectViewers()
{
    foreach (object item in viewerHolder.Items)
    {
        KinectDiagnosticViewer kinectViewer = item as KinectDiagnosticViewer;
        kinectViewer.Kinect = null;
    }
    viewerHolder.Items.Clear();
}
```

- First the Kinect for any open Viewers is set to null
- After looping through all open viewers and removing the Kinects, all Viewers are then removed from the UI

FindViewer

The *FindViewer* method is used to identify the KinectDiagnosticViewer, if any exists, associated with a given Kinect:

```
private KinectDiagnosticViewer FindViewer(Runtime runtime)
{
    return (from v in viewerHolder.Items.OfType<KinectDiagnosticViewer>() where
            Object.ReferenceEquals(v.Kinect, runtime) select v).FirstOrDefault();
}
```

- If no KinectDiagnosticViewer is found for the given Kinect this method returns null

Version0Manager.cs

The Version0Manager.cs file is the file that contains all the code to create and send the data packets to the DriverStation. It is located in the Networking\Protocols folder of the Kinect Server project.

Members and Constructors

The Version0Manager class has a number of members

- *HEARTBEAD_PERIOD_MS* is the maximum time between packets to the DriverStation in milliseconds
- *m_hostname* holds the destination to send the packets to
- *m_port* holds the port number to send the packets to
- *m_kinectVersion* holds the version string to be sent if the status indicates no error
- *m_kinectStatus* holds the current status string to be sent
- *m_udpClient* stores the UDPClient information
- *m_version0Packet* holds the actual packet being sent
- *m_heartbeatTimer* is a timer used to make sure packets are sent at a given minimum rate
- *m_started* indicates whether the Manager is running or not
- *m_gestureProcessor* holds the GestureProcessor to use to process the skeleton into joystick data

```
protected const int HEARTBEAT_PERIOD_MS = 450;
protected String m_hostname;
protected int m_port;
protected String m_kinectVersion;
protected String m_kinectStatus;
protected UdpClient m_udpClient;
protected Version0 m_version0Packet;
protected Timer m_heartbeatTimer;
protected bool m_started = false;
protected IGestureProcessor m_gestureProcessor;
```

The first constructor defined for Version0Manager takes 3 arguments, a **String** to use as the Version string, a **String** to use as the destination for the packets, and an **int** to use as the port to send the packets to:

```
public Version0Manager(String kinectVersion, String hostname, int port)
{
    m_kinectVersion = kinectVersion;
    m_kinectStatus = "No Kinect";
    m_hostname = hostname;
    m_port = port;
    m_udpClient = new UdpClient();
    m_version0Packet = new Version0();
    m_heartbeatTimer = new Timer(this.heartBeatExpired);
    m_heartbeatTimer.Change(HEARTBEAT_PERIOD_MS, HEARTBEAT_PERIOD_MS);
    m_started = true;
    m_gestureProcessor = new FIRSTGestureProcessor();
}
```

- This constructor initializes the network related variables using the parameters passed in
- The *m_HeartbeatTimer* is set up with the *heartBeatExpired* event handler attached to the timer expiration event and the expiration time set based on the period constant. The Gesture Processor is set to the default FIRSTGestureProcessor

The second constructor takes the same parameters as the first, plus an additional parameter defining the Gesture Processor to use:

```
public Version0Manager(IGestureProcessor gp, String kinectVersion, String hostname, int port)
   : this(kinectVersion,
         hostname,
         port)
{
   m_gestureProcessor = gp;
}
```

- This constructor can be used if you would like to write your own Gesture Processor completely from scratch implementing the IGestureProcessor interface without having to modify any of the code that sets up the Kinect or creates the actual packets.

Public API

Close

The *Close* method closes the UDP port and stops the heartbeat timer:

```
public void Close()
{
   m_heartbeatTimer.Change(Timeout.Infinite, Timeout.Infinite);
   m_udpClient.Close();
   m_started = false;
}
```

SetKinectStatus

The *SetKinectStatus* method sets the status string being sent with the packets based on the given status:

```
public void SetKinectStatus(MainWindow.ErrorCondition status)
{
   switch (status)
   {
      case MainWindow.ErrorCondition.None:
         m_kinectStatus = m_kinectVersion;
         break;
      case MainWindow.ErrorCondition.NoKinect:
         m_kinectStatus = "No Kinect";
         break;
      case MainWindow.ErrorCondition.NotReady:
         m_kinectStatus = "Not Ready";
         break;
      case MainWindow.ErrorCondition.NoPower:
         m_kinectStatus = "No Power";
         break;
      case MainWindow.ErrorCondition.KinectAppConflict:
         m_kinectStatus = "In Use";
         break;
      case MainWindow.ErrorCondition.EngConflict:
```

```
            m_kinectStatus = "Eng In Use"
            break;
        default:
            break;
    }
}
```

- The statuses passed to this function are members of the **ErrorCondition** enum defined in the **MainWindow**

Process Skeleton

The *ProcessSkeleton* method processes the given skeleton data and sends the resulting packet:

```
public void ProcessSkeleton(SkeletonFrame frame)
{
    sbyte[] nullAxis = new sbyte[6];

    lock (m_version0Packet)
    {
        m_version0Packet.PlayerCount.Set((byte)KinectUtils.CountTrackedSkeletons(frame.Skeletons));

        m_version0Packet.Flags.Set(0);      //Flags only accessible in the C++ API??
        m_version0Packet.FloorClipPlane.Set(frame.FloorClipPlane.X,
                                            frame.FloorClipPlane.Y,
                                            frame.FloorClipPlane.Z,
                                            frame.FloorClipPlane.W);
        m_version0Packet.GravityNormal.Set(frame.NormalToGravity.X,
                                            frame.NormalToGravity.Y,
                                            frame.NormalToGravity.Z);

        // Get the best skeleton
        SkeletonData s = KinectUtils.SelectBestSkeleton(frame.Skeletons);

        m_version0Packet.Quality.Set((byte)s.Quality);
        m_version0Packet.CenterOfMass.Set(s.Position.X,
                                          s.Position.Y,
                                          s.Position.Z);
        m_version0Packet.SkeletonTrackState.Set((uint) s.TrackingState);

        // Loop through joints; get tracking states and positions
        byte[] trackingStates = new byte[20];
        WritableVertex[] vertices = new WritableVertex[20];

        for (uint i = 0; i < s.Joints.Count; i++)
        {
            Joint j = s.Joints[(JointID)i];

            trackingStates[i] = (byte)(j.TrackingState);
            vertices[i] = new WritableVertex(j.Position.X,
                                             j.Position.Y,
```

```
                                    j.Position.Z);
            }
            m_version0Packet.SkeletonTrackingStates.Set(trackingStates);
            m_version0Packet.SkeletonJoints.Set(vertices);

            // Update Joysticks
            WritableJoystick[] sticks = new WritableJoystick[2] {
                m_version0Packet.Joystick1,
                m_version0Packet.Joystick2
            };
            if (m_version0Packet.PlayerCount.Get() != 0)       //Only process and send valid data if a player is detected
            {
                m_gestureProcessor.ProcessGestures(sticks, s);
            }
            else
            {
                m_version0Packet.Joystick1.Set(nullAxis, 0);
                m_version0Packet.Joystick2.Set(nullAxis, 0);
            }
        }

        send();
        m_heartbeatTimer.Change(HEARTBEAT_PERIOD_MS, HEARTBEAT_PERIOD_MS);
    }
```

- A lock is obtained on the packet in order to ensure that all data sent in a single packet comes from a single frame
- The PlayerCount field of the packet is set using the result of the *KinectUtils.CountTrackedSkeletons* method
- The Flags, FloorClipPlane, and GravityNormal fields of the packet are set directly using the data from the SkeletonFrame
- A single Skeleton is selected using the *KinectUtils.SelectBestSkeleton* method, this method returns the closest actively tracked skeleton
- The data from this skeleton is then used to set the Quality, CenterOfMass, SkeletonTrackState (overall state), SkeletonTrackingStates (individual joint states) and SkeletonJoints fields of the packet
- If any players are detected, joystick data is set using the designated GestureProcessor, of not the joystick data is set to all 0's.
- The completed packet is sent out
- The heartbeat timer is then reset

Private Methods

heartBeatExpired

The *heartBeatExpired* event handler is called when the Heartbeat timer expires. It is used to send a packet with 0 players and 0 for the joystick data in the event that SkeletonFrames are not being received, due to a disconnected Kinect or other malfunction:

```
    private void heartBeatExpired(Object stateinfo)
    {
        sbyte[] nullAxis = new sbyte[6];
```

```
        m_version0Packet.PlayerCount.Set(0);
        m_version0Packet.Joystick1.Set(nullAxis, 0);
        m_version0Packet.Joystick2.Set(nullAxis, 0);
        send();
    }
```

send

The *send* method is used to send the constructed packets:

```
    private void send()
    {
        if (!m_started)
            throw new InvalidOperationException("This Version0Manager has been closed and is no longer usable.");

        MemoryStream udpbuffer = new MemoryStream();
        NetworkOrderBinaryWriter writer = new NetworkOrderBinaryWriter(udpbuffer);

        lock (m_version0Packet)
        {
            m_version0Packet.VersionNumber.Set(m_kinectStatus);
            m_version0Packet.Serialize(writer);
        }

        m_udpClient.Send(udpbuffer.GetBuffer(), (int) udpbuffer.Length, m_hostname, m_port);
    }
```

- A lock is obtained on the packet to make sure that it is not changed during the operation, then the version string is set based on the Kinect status and the packet is serialized for network sending
- The packet is then sent to the previously specified destination and port

KinectUtils.cs

The KinectUtils.cs file contains a pair of methods used when processing the skeleton

CountTrackedSkeletons

The *CountTrackedSkeletons* method loops through the skeletons and counts how many are actively tracked (0-2):

```
public static ushort CountTrackedSkeletons(SkeletonData[] skeletons)
{
    ushort count = 0;

    foreach (SkeletonData skeleton in skeletons)
    {
        if (skeleton.TrackingState == SkeletonTrackingState.Tracked)
            count++;
    }
    return count;
}
```

- This function does not count the number of passively tracked skeletons (up to 4) which are identified by SkeletonTrackingState.PositionOnly

SelectBestSkeleton

The *SelectBestSkeleton* method is used to select which skeleton to use if multiple are present. It will always select one of the two actively tracked skeletons. This method compares the position of the two skeletons and chooses the closest one:

```
public static SkeletonData SelectBestSkeleton(SkeletonData[] skeletons)
{
    SkeletonData bestSkeleton = skeletons[0];

    foreach (SkeletonData skeleton in skeletons)
    {
        if (skeleton.TrackingState == SkeletonTrackingState.Tracked && bestSkeleton.TrackingState !=
            SkeletonTrackingState.Tracked || (skeleton.TrackingState == bestSkeleton.TrackingState &&
            skeleton.Position.Z < bestSkeleton.Position.Z))
          bestSkeleton = skeleton;
    }

    return bestSkeleton;
}
```

FIRSTGestureProcessor.cs

The processing to create joystick data from a skeleton using the default gestures is contained in the file FIRSTGestureProcessor.cs. The coordinate system used by the Kinect for Skeleton data has the Z-axis extending outward from the device toward the player (depth), the Y-axis extending from the top of the device (up) and the X-Axis extending from the side of the device (left from the device perspective, right from the player perspective). Helper functions and variable names in the FIRSTGestureProcessor are defined using this coordinate system

Members and Constructor

The FIRSTGestureProcessor class has no defined constructor; it relies on the default constructor. It does define a few constants and a series of delegate methods:

- Z_PLANE_TOLERANCE is used to define the acceptable distance along the z-axis between the shoulder and wrist for the arm to still be considered in a valid position
- ARM_MAX_ANGLE is used to define the positive side (arm up) of the valid range for the arm angle. This number is greater than 90 degrees as it is easy to reach beyond 90 with your arms above your head
- ARM_MIN_ANGLE is used to define the negative side (arm down) of the valid range for arm angle. This number is set to -90 so that moving your arm in front of your body is outside the valid range.
- CheckAngle is defined as a delegate. For more information on delegates see this page from Microsoft
- 5 delegate methods are defined with different angle checks for various gestures

```
public const double Z_PLANE_TOLERANCE = 0.3;

public const double ARM_MAX_ANGLE = 105;
public const double ARM_MIN_ANGLE = -90;

delegate bool CheckAngle(double angle);

CheckAngle IsLegForward = x => x < -110;
CheckAngle IsLegBackward = x => x > -80;
CheckAngle IsLegOut = x => x > -75;
CheckAngle IsHeadLeft = x => x > 98;
CheckAngle IsHeadRight = x => x < 82;
```

Helper Functions

RadToDeg

The *RadToDeg* method is a simple method to convert a given value in radians into degrees:

```
private double RadToDeg(double rad)
{
   return (rad * 180 ) / Math.PI;
}
```

AngleXY

This method returns the angle (in radians) of the vector pointing from Origin to Measured projected to the XY plane. If the mirrored parameter is true the vector is flipped about the Y-axis:

```
private double AngleXY(Vector origin, Vector measured, bool mirrored = false)
{
   return Math.Atan2(measured.Y - origin.Y, (mirrored) ? (origin.X - measured.X) : (measured.X –
      origin.X));
}
```

- The mirrored parameter is used to avoid the region where Atan2 is discontinuous

AngleXZ

The *AngleYZ* method is very similar to the *AngleXY* method with the projection done to the YZ plane instead of the XY plane:

```
private double AngleYZ(Vector origin, Vector measured, bool mirrored = false)
{
   return Math.Atan2(measured.Y - origin.Y, (mirrored) ? (origin.Z - measured.Z) : (measured.Z –
       origin.Z));
}
```

InSameZPlane

The *InSameZPlane* returns whether the two points are in approximately the same XY plane along the Z-axis. The tolerance parameter determines how close the Z coordinates of the points have to be to return True:

```
private bool InSameZPlane(Vector origin, Vector measured, double tolerance)
{
   return Math.Abs(measured.Z - origin.Z) < tolerance;
}
```

CoerceToRange

The *CoerceToRange* method takes an input, an input range, and an output range, and uses them to scale and constrain the input to the output range:

```
private double CoerceToRange(double input, double inputMin, double inputMax, double outputMin,
     double outputMax)
{
   double inputCenter = Math.Abs(inputMax - inputMin) / 2 + inputMin;
```

```
        double outputCenter = Math.Abs(outputMax - outputMin) / 2 + outputMin;

        double scale = (outputMax - outputMin) / (inputMax - inputMin);

        double result = (input + -inputCenter) * scale + outputCenter;

        return Math.Max(Math.Min(result, outputMax), outputMin);
    }
```

- First the center of the input and output ranges is calculated
- Next the scale factor between the two ranges is calculated
- The input range is shifted to 0; the input is scaled by the calculated factor, and then shifted to match the outputCenter.
- The result is then constrained to the output range (the result of the transformation can exceed the output range if the input is outside the input range)

Gesture Processing

ProcessGestures

The *ProcessGestures* method is responsible for doing the actual processing of the skeleton data into joystick data using the default gestures. After checking that the Joysticks are in a valid state and initializing some variables, the Axis data is created using the position of the left and right arms

```
public void ProcessGestures(Networking.WritableElements.WritableJoystick[] joy,
Microsoft.Research.Kinect.Nui.SkeletonData skeleton)
      {

         // Check edge cases
         if (joy == null || joy.Length < 2 || joy[0] == null || joy[1] == null)
            return;

         sbyte[] leftAxis = new sbyte[6];
         sbyte[] rightAxis = new sbyte[6];
         sbyte[] nullAxis = new sbyte[6];
         bool dataWithinExpectedRange;
         ushort buttons = 0;

         double leftAngle = RadToDeg(AngleXY(skeleton.Joints[JointID.ShoulderLeft].Position,
                                    skeleton.Joints[JointID.WristLeft].Position,
                                    true));

         double rightAngle = RadToDeg(AngleXY(skeleton.Joints[JointID.ShoulderRight].Position,
                                     skeleton.Joints[JointID.WristRight].Position));

         dataWithinExpectedRange = leftAngle < ARM_MAX_ANGLE && leftAngle > ARM_MIN_ANGLE &&
                              rightAngle < ARM_MAX_ANGLE && rightAngle > ARM_MIN_ANGLE;

         double leftYAxis = CoerceToRange(leftAngle,
                                    -70,
                                    70,
                                    -127,
                                    128);

         double rightYAxis = CoerceToRange(rightAngle,
                                    -70,
                                    70,
                                    -127,
                                    128);

         dataWithinExpectedRange = dataWithinExpectedRange &&
                              InSameZPlane(skeleton.Joints[JointID.ShoulderLeft].Position,
                                    skeleton.Joints[JointID.WristLeft].Position,
                                    Z_PLANE_TOLERANCE) &&
                              InSameZPlane(skeleton.Joints[JointID.ShoulderRight].Position,
                                    skeleton.Joints[JointID.WristRight].Position,
                                    Z_PLANE_TOLERANCE);
```

- The shoulder joint is used as the origin joint and the wrist joint as the measured joint for each arm so the vector being measured points from shoulder to wrist. The left measurement is mirrored in order to avoid the discontinuous region of the Atan2 function.
- A check is done to see if both arm angles are within the valid range
- The arm angles are then scaled and coerced into the range of a Joystick Axis in the FRC system (Note: the range here is -127 to 128 because the data is inverted later, the actual joystick range is -128 to 127)
- Both arms are then checked to make sure they approximately straight out

Next the button values are determined:

```
// Head buttons
double headAngle = RadToDeg(AngleXY(skeleton.Joints[JointID.ShoulderCenter].Position,
                            skeleton.Joints[JointID.Head].Position));
if (IsHeadRight(headAngle))
   buttons |= (ushort) WritableJoystick.Buttons.Btn1;
if (IsHeadLeft(headAngle))
   buttons |= (ushort)WritableJoystick.Buttons.Btn2;


// Right Leg XY Button
double rightLegAngle = RadToDeg(AngleXY(skeleton.Joints[JointID.HipRight].Position,
                             skeleton.Joints[JointID.AnkleRight].Position));
if (IsLegOut(rightLegAngle))
   buttons |= (ushort)WritableJoystick.Buttons.Btn3;

// Left Leg XY Button
double leftLegAngle = RadToDeg(AngleXY(skeleton.Joints[JointID.HipLeft].Position,
                            skeleton.Joints[JointID.AnkleLeft].Position,
                            true));
if (IsLegOut(leftLegAngle))
   buttons |= (ushort)WritableJoystick.Buttons.Btn4;

// Right Leg YZ Buttons
double rightLegYZ = RadToDeg(AngleYZ(skeleton.Joints[JointID.HipRight].Position,
                            skeleton.Joints[JointID.AnkleRight].Position));
if (IsLegForward(rightLegYZ))
   buttons |= (ushort)WritableJoystick.Buttons.Btn5;
if (IsLegBackward(rightLegYZ))
   buttons |= (ushort)WritableJoystick.Buttons.Btn6;

// Left Leg YZ Buttons
double leftLegYZ = RadToDeg(AngleYZ(skeleton.Joints[JointID.HipLeft].Position,
                            skeleton.Joints[JointID.AnkleLeft].Position));
if (IsLegForward(leftLegYZ))
   buttons |= (ushort)WritableJoystick.Buttons.Btn7;
if (IsLegBackward(leftLegYZ))
   buttons |= (ushort)WritableJoystick.Buttons.Btn8;
```

- Angles for each joint pair are calculated (again mirroring the left side in XY calculations) and then compared to predetermined values using the delegate functions

If appropriate the axis and button data is written to the Joystick:

```
        if (dataWithinExpectedRange)
        {
            // Invert joystick axis to match a real joystick
            // (pushing away creates negative values)
            leftAxis[(uint)WritableJoystick.Axis.Y] = (sbyte) -leftYAxis;
            rightAxis[(uint)WritableJoystick.Axis.Y] = (sbyte) -rightYAxis;

            //Use "Button 9" as Kinect control enabled signal
            buttons |= (ushort)WritableJoystick.Buttons.Btn9;

            joy[0].Set(leftAxis, buttons);
            joy[1].Set(rightAxis, buttons);
        }
        else
        {
            joy[0].Set(nullAxis, 0);
            joy[1].Set(nullAxis, 0);

        }
    }
```

- If both arm angles are within the valid range and both arms are approximately straight out, Button 9 is set to True to and the axis and button values are written to the joystick
- If any of those checks failed, all 0's are written to the Joysticks

Modifying or Creating Gestures

Teams looking to modify the default gestures or add new ones to the existing Kinect Server have a few options:

1. Minor modifications such as angle thresholds or ranges can be made by changing the values in this file
2. Gestures can be added or replaced by modifying and/or adding to the code in this file.
3. An entirely new gesture processor can be defined by implementing the IGestureProcessor interface and changing MainWindow.xaml.cs to call the alternate Version0Manager constructor using your gesture processor.

Minor Modifications

To make minor modifications to the existing gestures such as changing the threshold angles for the buttons or the angle ranges for the arms, you can change the appropriate numbers then compile the new Kinect Server and place it in the FRC Kinect Server directory.

Adding or Replacing Gestures

To add additional gestures or replace existing default gestures you will need to write to the appropriate joystick elements.

To set a joystick axis use a line similar to this one:

```
leftAxis[(uint)WritableJoystick.Axis.Y] = (sbyte)What you want the axis to be
```

leftAxis is used for KinectStick1 and **rightAxis** is used for KinectStick2. Replace the "**Y**" with the axis you wish to use. Valid axis options are X, Y, Z, Twist, Throttle, and Custom. The right side of the assignment is cast to a signed byte so it must be in the range (-128, 127) or unexpected behavior will result. -128 will correspond with -1 in your robot program and 127 will correspond with +1.

To set a button use a line similar to this one

```
buttons |= (ushort)WritableJoystick.Buttons.Btn#;
```

Replace '#' with the number of the button you wish to set. Valid buttons are 1-12, with 1-9 taken up by the default gestures.

Your axes and buttons should all be written to the **leftAxis**, **rightAxis** and **buttons** variables before the line:

```
joy[0].Set(leftAxis, buttons);
```

If you are modifying or replacing the default arm gestures you may have to modify or remove the checks used with the **dataWithinExpectedRange** variable. **For safety purposes, it is strongly recommended to modify the check to define your own valid positioning, removing the check entirely is NOT RECOMMENDED.**

The existing gesture code can be used as an example of how to access Joint data, for a complete reference of the available skeleton data see the Microsoft Kinect for Windows SDK API Reference. This document can be found by going to Start->All-Programs->Microsoft Kinect 1.0 Beta2 SDK->Kinect SDK Documentation. Once this help file is open select Managed Code Reference->NUI API.

PART 3— Resources

Learn Visual C#

WPF Windows Overview

Official Kinect for Windows Site

Kinect SDK Quickstart Video Tutorials

Coding4Fun Kinect Toolkit

Coding4Fun Kinect Projects