KINECT
for Windows

Kinect for Windows API V2

John Elsbree

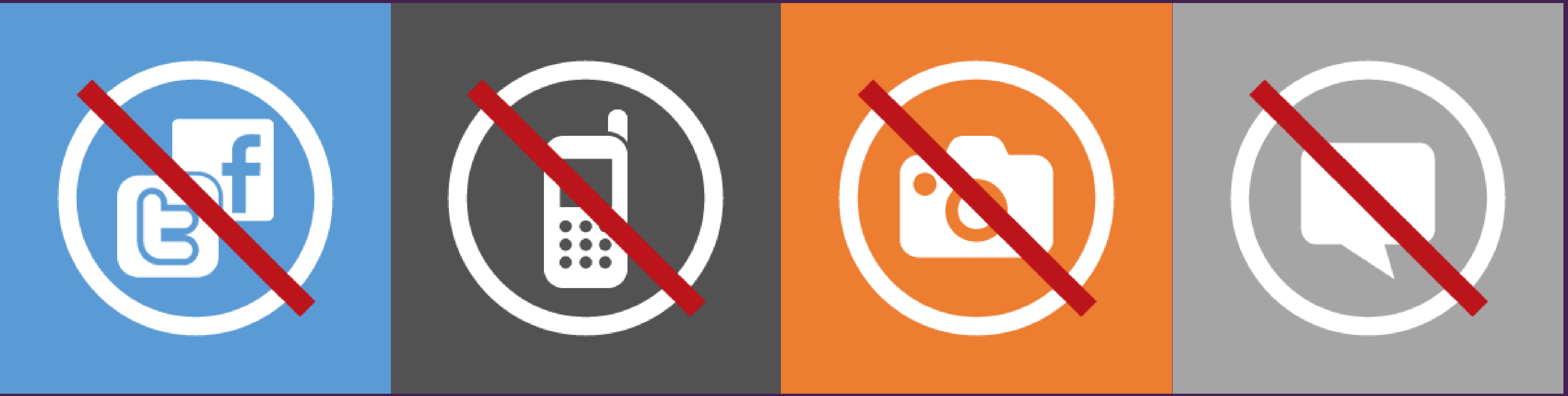
Principal Software Development Engineer
Kinect for Windows Program

18 November 2013

Microsoft

Everything is under NDA

Unless otherwise stated.



Agenda

- V2 Kinect sensor
- API overview
  - Influences and style
  - Differences from V1
- API features
  - Sensor lifecycle
  - Data sources
  - Frame synchronization
  - Coordinate mapping
- Porting guidance
- Not covered in this session: speech, face tracking, interactions

V2 Kinect sensor

Sensor differences from V1 (briefly)

- One color camera resolution (1920x1080), frame rate (30 Hz)
- One depth/IR camera resolution (512x424), frame rate (30 Hz)
- Depth range from 0.5 to 4.5 m
- Clean infrared frames
- Can use infrared and color cameras simultaneously
- No tilt motor (wider field of view)

API overview

COM and .NET

- V2 API still has both COM and .NET flavors
- Much more similar *to each other* than they were in V1
- COM and .NET each have unique patterns for expressing:
  - Events
  - Buffers
  - Collections
- V1 code *cannot* just be recompiled for V2; some changes will be needed
- Existing .NET code will need fewer changes than COM code

Influences and style

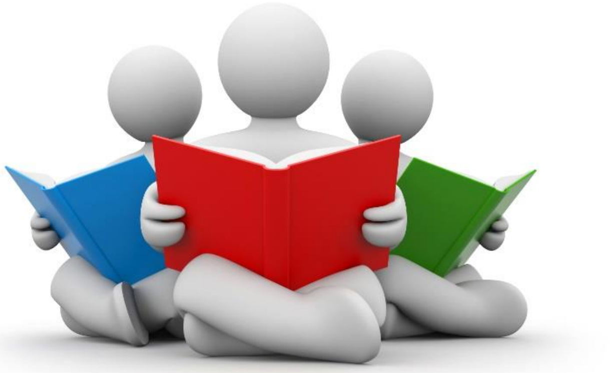
- New API is influenced by WinRT
- Xbox One has a WinRT API for Kinect
- Kinect for Windows V2 API is ***not*** a WinRT API (yet)
- “Shape” of V2 COM and .NET APIs identical to Xbox One’s WinRT API
  - Same types, methods, etc.
- New Kinect API uses WinRT idioms
  - Sources/readers
  - Events
  - Collections

API differences from V1

- All resolutions and frame rates are constant
- Can use color and infrared simultaneously
- No separate “near” mode for depth (range is 0.5 to 4.5 m)
- “Skeleton” → “Body”
  - 6 fully-tracked bodies
  - More joints per body
  - More features: expressions, actions, lean, …
  - No separate “seated” mode (both seated and standing can be tracked)
- Depth and body index delivered as separate frame types
- More audio beam features

Source/Reader pattern

- V1: Once a frame was retrieved from a stream by polling, it was gone forever
- V2: Multiple reader instances can independently poll the same source for frames, without interference
- Readers can be paused/resumed independently
- Enables more componentized applications



Source/Reader pattern

- V1: sensor → stream → frame → data
  - Stream – 1 of each type per sensor

- V2: sensor → source → reader → frame → data
  - Source – 1 of each type per sensor
  - Reader – many per source

Events

- COM:
  - HRESULT Subscribe*EventName*(WAITABLE_HANDLE *waitableHandle);
  - HRESULT Unsubscribe*EventName*(WAITABLE_HANDLE waitableHandle);
  - HRESULT Get*EventName*EventData(WAITABLE_HANDLE waitableHandle, I*EventName*EventArgs **eventData);
- .NET:
  - public event EventHandler<*EventName*EventArgs> *EventName*;

Buffers

- COM:
  - HRESULT CopyFrameDataToArray(UINT capacity, BYTE* buffer);

- .NET:
  - public void CopyFrameDataToArray(byte[] frameData);
  - public void CopyFrameDataToBuffer(uint bufferSize, IntPtr buffer);

Collections

- COM:
  - `interface IKinectSensorCollection`
- .NET:

  -
    ```
    class KinectSensorCollection :
      IReadOnlyList<KinectSensor>,
      INotifyCollectionChanged
    ```

API Features

Sensor lifecycle

- Lifecycle
  - Find a KinectSensor object
  - Open it
  - Use it
  - Close it
- Sensor unplugged:
  - KinectSensor object remains valid
  - Your code still runs
  - No frames arrive
  - KinectSensor.IsAvailable tells you if it’s actually there

Sensor lifecycle – One sensor

```
this.sensor = KinectSensor.Default;
this.sensor.Open();
…
this.sensor.Close();
```

Sensor lifecycle – Multiple sensors

- **NOTE**: Not yet implemented in Tech Preview

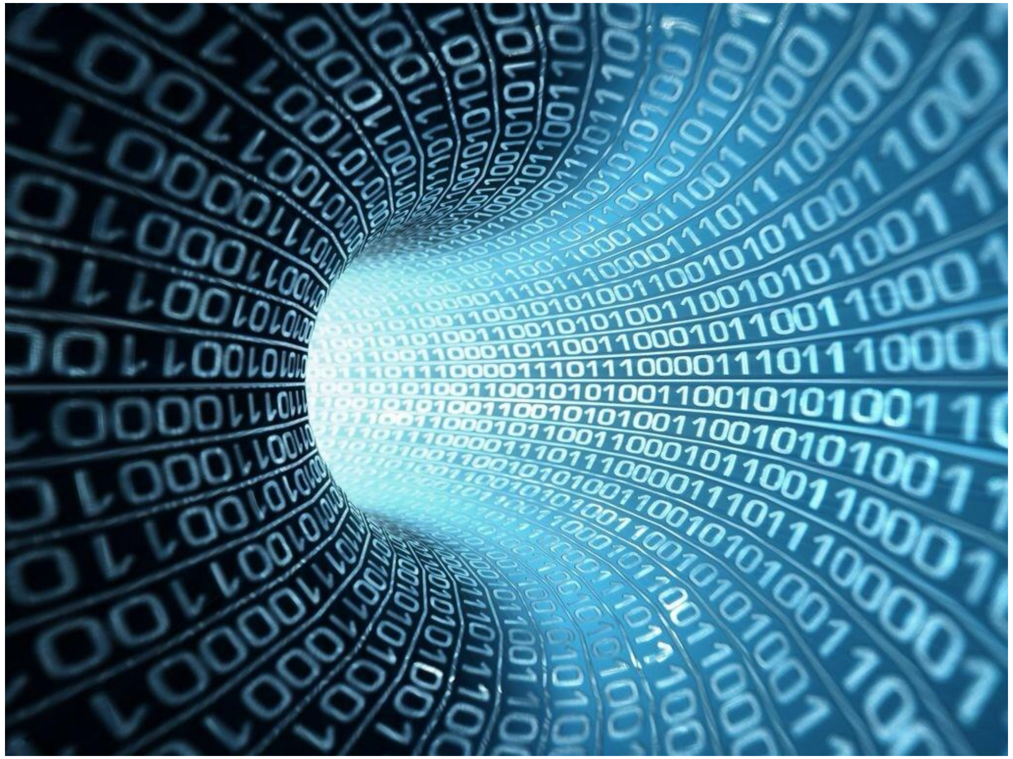
```
foreach (KinectSensor sensor in KinectSensor.Sensors)
{
  string sensorId = sensor.UniqueKinectId;
  this.sensors.Add(sensorId, sensor);

  sensor.Open();
  …
}
…

foreach (KinectSensor sensor in this.sensors.Values)
{
  sensor.Close();
}
```

Data sources

- Infrared
- Color
- Depth
- Body index
- Body
- Audio



Infrared

- The simplest sources to use
- Two forms: Infrared and LongExposureInfrared
  - Infrared is a single frame
  - LongExposureInfrared is sum of 3 IR frames (higher signal:noise ratio, but more motion blur)
- Examples that follow use single-frame Infrared
  - To use long-exposure infrared, just "Infrared" → "LongExposureInfrared"
- Frame data is 2 bytes per pixel: 16-bit IR intensity value (same as V1)

Infrared – Initialization (.NET)

```
// Allocate a buffer
FrameDescription frameDesc =
  this.sensor.InfraredFrameSource.FrameDescription;
this.infraredData = new ushort[frameDesc.LengthInPixels];

// Open a reader and subscribe to frame arrival events
this.infraredReader = this.sensor.InfraredFrameSource.OpenReader();
this.infraredReader.FrameArrived += InfraredFrameArrived;
```

Aside – FrameDescription

```
public sealed class FrameDescription
{
    public int Width { get; }
    public int Height { get; }
    public float HorizontalFieldOfView { get; }
    public float VerticalFieldOfView { get; }
    public float DiagonalFieldOfView { get; }
    public uint LengthInPixels { get; }
    public uint BytesPerPixel { get; }
}
```

Infrared – Event handler (.NET)

```
private void InfraredFrameArrived(
  object sender,
  InfraredFrameArrivedEventArgs e)
{
  bool processFrame = false;

  // Acquire the frame
  using (InfraredFrame frame = e.FrameReference.AcquireFrame())
  {
    if (null != frame)
    {
      // Copy frame's data to our buffer
      frame.CopyFrameDataToArray(this.infraredData);
      processFrame = true;
    }
  }

  if (processFrame) { ProcessInfraredData(this.infraredData); }
}
```

Aside – FrameReference

- FrameReference.RelativeTime property is a timestamp for the frame it represents
- By the time the app calls FrameReference.AcquireFrame, the frame might already have “expired”
  - A newer frame has already taken its place
  - AcquireFrame returns null
  - Caller always needs to check

Infrared – Initialization (COM)

```
// Access the frame source
IInfraredFrameSource* pSource = nullptr;
this->_pSensor->get_InfraredFrameSource(&pSource);

// Allocate a buffer
IFrameDescription* pFrameDesc = nullptr;
pSource->get_FrameDescription(&pFrameDesc);
pFrameDesc->get_LengthInPixels(&this->_lengthInPixels);
pFrameDesc->Release();
this->_pInfraredData = new UINT16[this->_lengthInPixels];

// Open a reader and subscribe to frame arrival events
pSource->OpenReader(&this->_pInfraredReader);
this->_pInfraredReader->SubscribeFrameArrived(&this->_hInfraredEvent);
pSource->Release();
```

Infrared – Event dispatch (COM)

```
while (…)
{
  HANDLE handles[] = { reinterpret_cast<HANDLE>(this->_hInfraredEvent), … };
  switch (WaitForMultipleObjects(_countof(handles), handles, …);
  {
  case WAIT_OBJECT_0:
    {
      IInfraredFrameArrivedEventArgs* pArgs = nullptr;
      this->_pInfraredReader->GetFrameArrivedEventData(this->_hInfraredEvent,
        &pArgs);
      InfraredFrameArrived(pArgs);
      pArgs->Release();
    }
    break;
    …
  }
}
```

Infrared – Event handler (COM)

```
void MyClass::InfraredFrameArrived(IInfraredFrameArrivedEventArgs* pArgs)
{
  IInfraredFrameReference* pFrameReference = nullptr;
  pArgs->get_FrameReference(&pFrameReference);

  // Acquire the frame
  bool processFrame = false;
  IInfraredFrame* pFrame = nullptr;
  if (SUCCEEDED(pFrameReference->AcquireFrame(&pFrame)))
  {
    // Copy frame's data to our buffer
    pFrame->CopyFrameDataToArray(this->_lengthInPixels, this->_pInfraredData);
    processFrame = true;
    pFrame->Release();
  }
  pFrameReference->Release();

  if (processFrame) { ProcessInfraredData(this->_lengthInPixels, this->_pInfraredData); }
}
```

Infrared – Other features (.NET)

- Polling (instead of events)

```
InfraredFrame frame =
  this.infraredReader.AcquireLatestFrame();
```

- Pause/resume

```
this.infraredReader.IsPaused = true;
```

- Raw buffer access

```
int size;
IntPtr buffer;
frame.AccessUnderlyingBuffer(out size, out buffer);
unsafe { ushort* bufferData = (ushort*)buffer; … }
```

- Timestamp

```
long timestamp = frame.RelativeTime;
```

Color

- Color frames have multiple possible formats (RGBA, BGRA, YUY2, …)
- Frames come from the sensor in a default raw format
  - YUY2 for now (Tech Preview), but may be different in the future
- Frame data can be:
  - Accessed in its raw format, or
  - Converted to another format (at slightly higher cost)
- Underlying buffer access is available only for the raw format
- Buffer is array of bytes, but typically multiple bytes per pixel (how many depends on format)

Color – Raw format

```
FrameDescription frameDesc =
  this.sensor.ColorFrameSource.FrameDescription;
this.colorData =
  new byte[frameDesc.LengthInPixels * frameDesc.BytesPerPixel];

ColorImageFormat rawColorFormat = ColorImageFormat.None;
using (ColorFrame frame = e.FrameReference.AcquireFrame())
{
  if (null != frame)
  {
    rawColorFormat = frame.RawColorImageFormat;
    frame.CopyRawFrameDataToArray(this.colorData);
  }
}
switch (rawColorFormat) …
```

Color – Format conversion

```
FrameDescription frameDesc =
  this.sensor.ColorSource.CreateFrameDescription(ColorImageFormat.Bgra);
this.colorData =
  new byte[frameDesc.LengthInPixels * frameDesc.BytesPerPixel];


using (ColorFrame frame = e.FrameReference.AcquireFrame())
{
  if (null != frame)
  {
    frame.CopyConvertedFrameDataToArray(
      this.colorData,
      ColorImageFormat.Bgra);
  }
}
```

Color – Convert directly to WriteableBitmap

```
// Determine dimensions from frame description
FrameDescription frameDesc = colorFrame.CreateFrameDescription(ColorImageFormat.Bgra);
int width = frameDesc.Width;
int height = frameDesc.Height;
uint bufferLength = (uint)(frameDesc.LengthInPixels * frameDesc.BytesPerPixel);

// Create bitmap if needed
if (null == this.colorBitmap)
{
  this.colorBitmap = new WriteableBitmap(width, height, 96, 96, PixelFormats.Bgra32, null);
}

// Convert color data directly into the bitmap
this.colorBitmap.Lock();
colorFrame.CopyConvertedFrameDataToBuffer(
  bufferLength, this.colorBitmap.BackBuffer, ColorImageFormat.Bgra);
this.colorBitmap.AddDirtyRect(new Int32Rect(0, 0, width, height));
this.colorBitmap.Unlock();
```

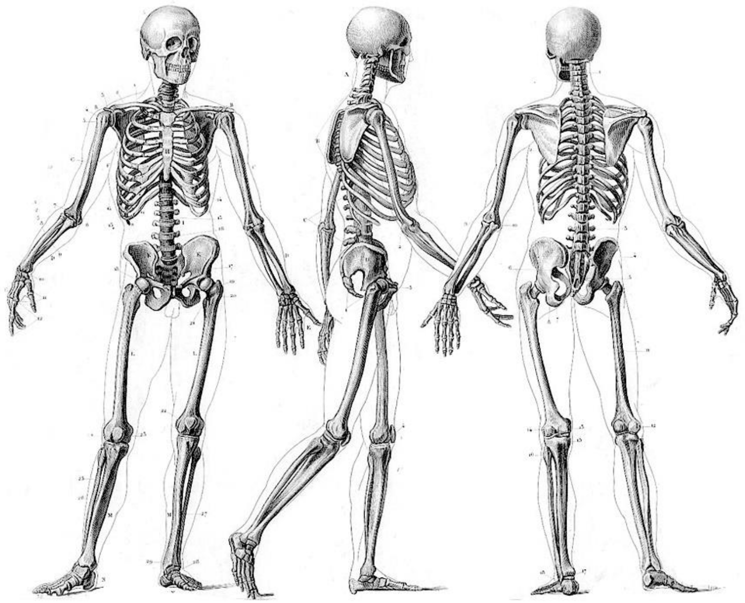
Depth

- Frame data is 2 bytes per pixel: 16-bit distance in millimeters
- No “PlayerIndex” – use BodyIndex source instead
- API nearly identical to Infrared
- Two additional properties:
  - DepthMinReliableDistance
  - DepthMaxReliableDistance
- Known bug in Tech Preview: DepthMaxReliableDistance is 4000, but should be 4500 (depth values up to 4500 mm are actually returned)

Body index

- Frame data is 1 byte per pixel: index of the body, as determined by body tracking
- Resolution is same as depth
- Pixel values (different from V1)
  - -1: No body at this pixel
  - 0 to 5: Index of the corresponding body, as tracked by the body source
  - All other values: not used
- Apart from pixel value size, API nearly identical to Infrared

Body



- Formerly known (in V1) as Skeleton
- Frame data is an array of Body objects
- Many new features in V2
  - More joints (neck, thumbs, hand tips)
  - Hand states (open, closed, “lasso”)
  - Activities (eye closed, mouth open, mouth motion, looking away)
  - Appearance (wearing glasses)
  - Level of user engagement
  - Facial expressions (happy, neutral)
  - Lean direction (2D vector, “human joystick”)

Body - Initialization

- Instead of FrameDescription, we have BodyCount

```
// Allocate a buffer of bodies
this.infraredData =
  new Body[this.sensor.BodyFrameSource.BodyCount];

// Open a reader and subscribe to frame arrival events
this.bodyReader = this.sensor.BodySource.OpenReader();
this.bodyReader.FrameArrived += BodyFrameArrived;
```

Body – Event handler

- Instead of CopyFrameDataToArray, we have GetAndRefreshBodyData

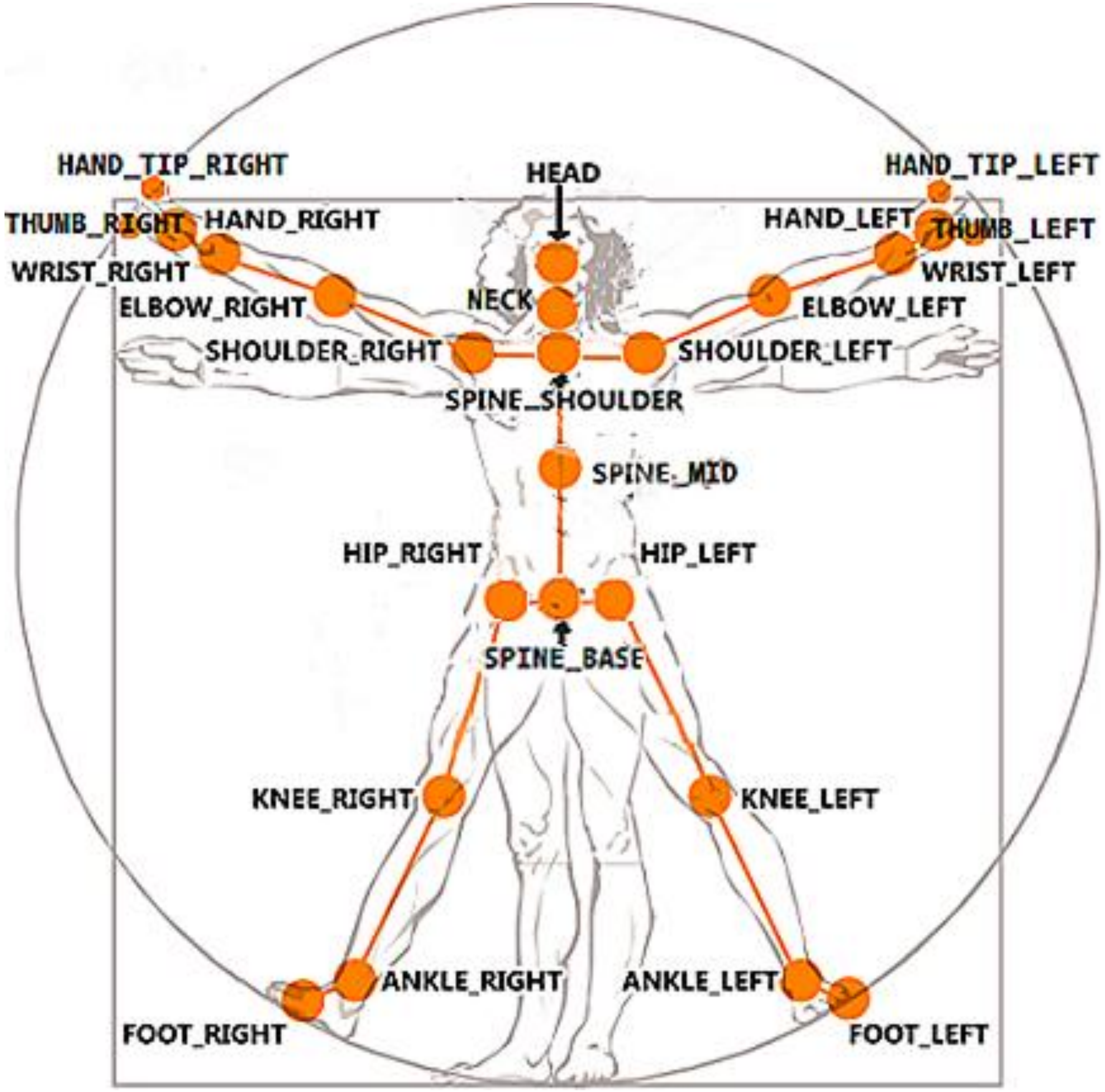
```
// Acquire the frame
using (BodyFrame frame = e.FrameReference.AcquireFrame())
{
  if (null != frame)
  {
    // Copy frame's data to our buffer
    frame.GetAndRefreshBodyData(this.bodyData);
  }
}
```

- Frame also has BodyCount and FloorClipPlane properties

Body – GetAndRefreshBodyData

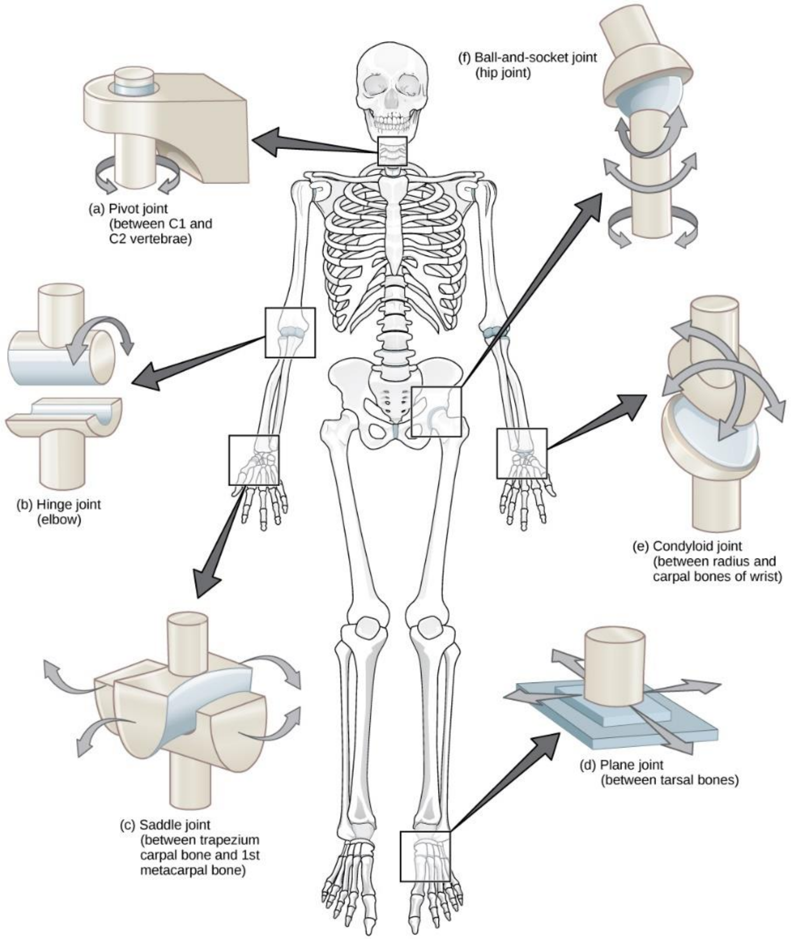
- Designed to minimize per-frame allocations of new objects
- Your responsibility: pass a Body array of the appropriate length (BodyCount)
- If an array element is null:
  - New Body is created and populated with data, and
  - Stored in the array
- If an array element is non-null:
  - Existing Body instance is reused, and
  - Content of the Body is overwritten with data for the new frame
- To retain a Body after its frame is disposed:
  - Keep a reference to it, and
  - Replace it in the array with null

Body - Joints


HAND_TIP_RIGHT
HEAD
HAND_TIP_LEFT
THUMB_RIGHT
HAND_RIGHT
HAND_LEFT
THUMB_LEFT
WRIST_RIGHT
WRIST_LEFT
NECK
ELBOW_RIGHT
ELBOW_LEFT
SHOULDER_RIGHT
SHOULDER_LEFT
SPINE_SHOULDER
SPINE_MID
HIP_RIGHT
HIP_LEFT
SPINE_BASE
KNEE_RIGHT
KNEE_LEFT
ANKLE_RIGHT
ANKLE_LEFT
FOOT_RIGHT
FOOT_LEFT

Body – Joints positions and orientations

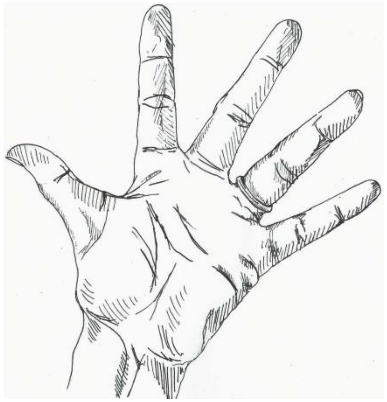
- Two dictionaries, each keyed by JointType:
  - Joints: tracking state and 3D position
  - Orientation: 3D orientation, specified as a quaternion
- Joint tracking state may be:
  - Not tracked
  - Inferred
  - Tracked



Body – Joint positions and orientations

```
foreach (Body body in this.bodyData)
{
  if (body.IsTracked)
  {
    foreach (Joint joint in body.Joints.Values)
    {
      if (joint.TrackingState != TrackingState.NotTracked)
      {
        CameraSpacePoint point = joint.Position;
        Vector4 rotation = body.JointOrientations[joint.JointType];
        …
      }
    }
  }
}
```

Body – Hand states



- Two properties: HandLeftState, HandRightState
  - Unknown
  - Not tracked
  - Open
  - Closed
  - Lasso
- Confidence properties: HandLeftConfidence, HandRightConfidence
  - High
  - Low
- Hand state tracking limited to 2 bodies at a time
  - BodyFrameSource.OverrideHandTracking lets you choose which bodies
  - Limit *may* be increased in the future

Body – Activities, Appearance, Expressions

- Activities
  - EyeLeftClosed
  - EyeRightClosed
  - MouthOpen
  - MouthMoved
  - LookingAway
- Appearance
  - WearingGlasses
- Expressions
  - Happy
  - Neutral

Body – Activities, Appearance, Expressions

- Three dictionaries
  - Key is a state type
  - Value indicates probability of that state (Unknown, No, Maybe, Yes)
- API may be extended in the future, by adding new state keys

```
bool isMouthClosed =
  (body.Activities[Activity.MouthOpen] == DetectionResult.No);

bool isWearingGlasses =
  (body.Appearance[Appearance.WearingGlasses] == DetectionResult.Yes;

bool isPossiblyHappy =
  (body.Expressions[Expression.Happy] >= DetectionResult.Maybe);
```

Body – Other properties

- Engaged

```
bool isEngaged = (body.Engaged == DetectionResult.Yes);
```

- Lean (body as a 2D joystick)

```
if (body.LeanTrackingState == TrackingState.tracked)
{
  float leanLeftRight = body.Lean.X;
  float leanForwardBack = body.Lean.Y;
}
```

- TrackingId: unique 64-bit ID assigned to each new body
- ClippedEdges: which edges of the field-of-view are clipping the body

Audio

- **NOTE**: Not yet implemented in Tech Preview
- Audio beam
  - A steerable “cone” of focus for audio
  - May be automatically or manually aimed
  - Kinect audio source API can support multiple beams
- Audio beam frame
  - Contains audio samples captured for a beam over a specific interval of time
  - Synchronized frames for all beams available as a group, on each event

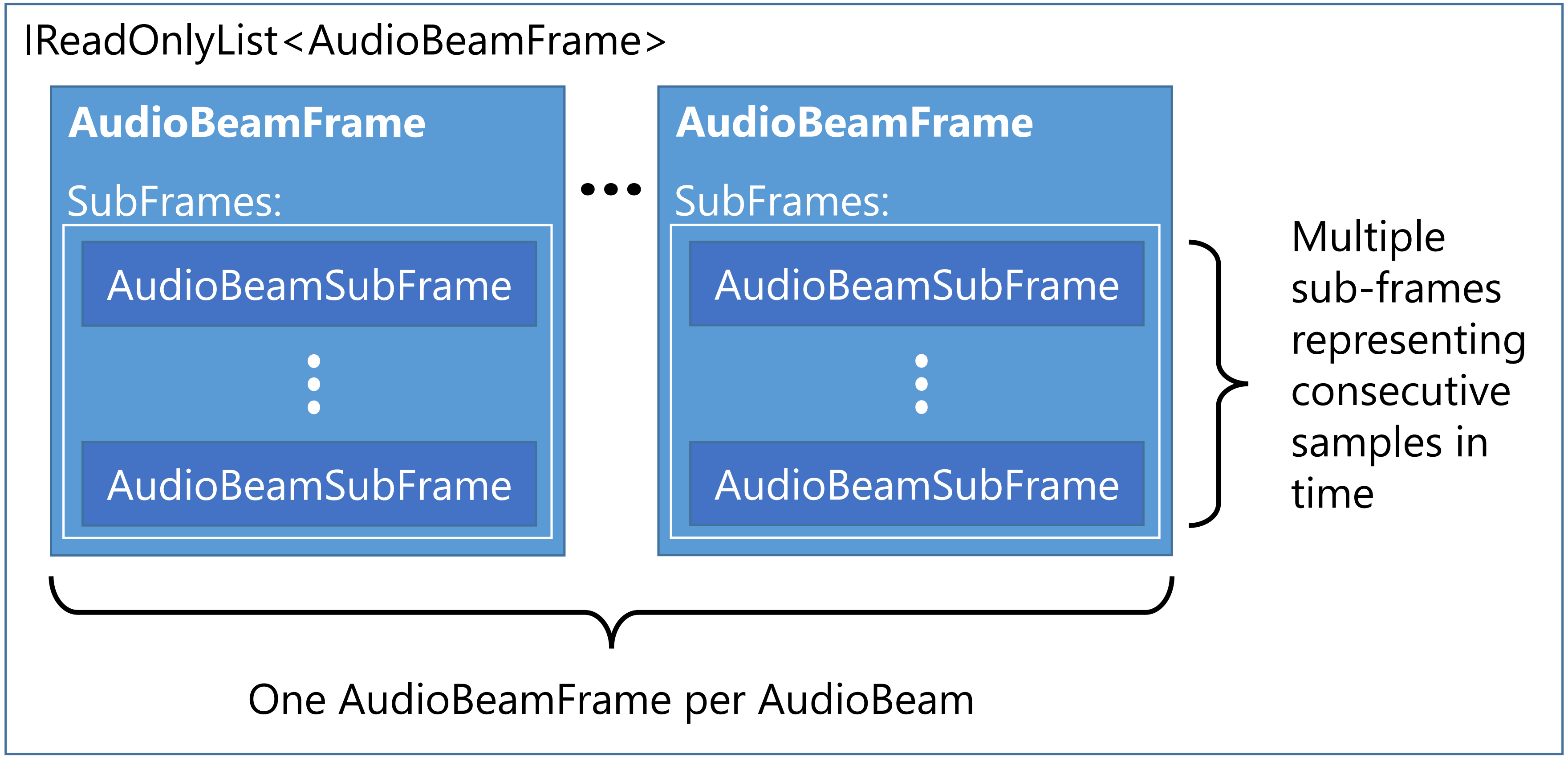
Audio – Single beam – Initialization

```
// Allocate a buffer
this.audioData = new
  byte[this.sensor.AudioSource.SubFrameLengthInBytes];

// Open a reader and subscribe to frame arrival events
this.audioReader = this.sensor.AudioSource.OpenReader();
this.audioReader.FrameArrived += AudioFrameArrived;
```

Audio – Beam frames

AudioBeamFrameReference.AcquireBeamFrames returns a list of AudioBeamFrames



IReadOnlyList<AudioBeamFrame>

AudioBeamFrame

SubFrames:

AudioBeamSubFrame

⋮

AudioBeamSubFrame

…

AudioBeamFrame

SubFrames:

AudioBeamSubFrame

⋮

AudioBeamSubFrame

Multiple sub-frames representing consecutive samples in time

One AudioBeamFrame per AudioBeam

Audio – Single beam – Event handler

```
void AudioFrameArrived(object sender, AudioBeamFrameArrivedEventArgs e)
{
  using (AudioBeamFrame beamFrame =
    e.FrameReference.AcquireBeamFrames().FirstOrDefault())
  {
    if (beamFrame != null)
    {
      foreach (AudioBeamSubFrame subFrame in beamFrame.SubFrames)
      {
        subFrame.CopyFrameDataToArray(this.audioData);
        float beamAngle = subFrame.BeamAngle;
        float beamAngleConfidence = subFrame.BeamAngleConfidence;
        long timestamp = subFrame.RelativeTime;
        subFrame.Dispose();

        ProcessAudioData(this.audioData, timestamp, beamAngle, beamAngleConfidence);
      }
    }
  }
}
```

Audio – Body correlation

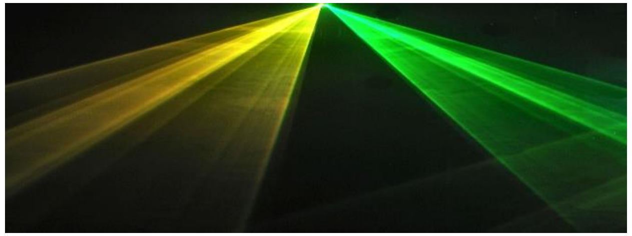
- Identifies which body (or bodies) are in the path of a beam

```
foreach (AudioBeamSubFrame subFrame in beamFrame.SubFrames)
{
  subFrame.CopyFrameDataToArray(this.audioData);
  long timestamp = subFrame.RelativeTime;

  foreach (AudioBodyCorrelation audioBody in
    subFrame.AudioBodyCorrelations)
  {
    ProcessAudioData(
      this.audioData, timestamp, audioBody.BodyTrackingId);
  }

  subFrame.Dispose();
}
```

Audio – Beam steering



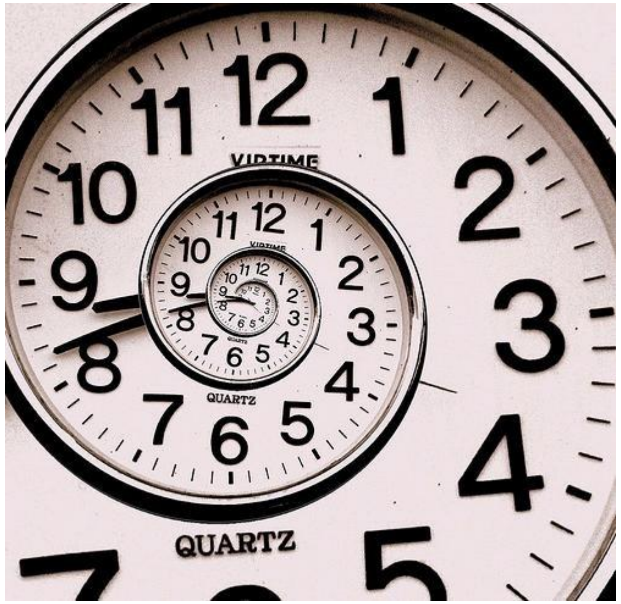
```
private int ManuallyAimAudioBeams(float[] targetAngles)
{
  int i = 0;
  foreach (AudioBeam beam in
    this.sensor.AudioSource.AudioBeams)
  {
    if (i >= targetAngles.Length) { break; }

    beam.AudioBeamMode = AudioBeamMode.Manual;
    beam.BeamAngle = targetAngles[i++];
  }

  return i;  // number of beams that were actually aimed
}
```

Frame synchronization

- Open a MultiSourceFrameReader, indicating which sources you want
- When a matched set of frames is ready, the event fires
- MultiSourceFrame contains references to each of the matched frames
- A more general-purpose form of the AllFramesReady event in V1



MultiSourceFrameReader - Initialization

```
// Open a multi-source reader
this.multiReader =
  this.sensor.OpenMultiSourceFrameReader(
    FrameSourceTypes.Color |
    FrameSourceTypes.Depth |
    FrameSourceTypes.Body |
    FrameSourceTypes.BodyIndex);

// Subscribe to frame events
this.multiReader.MultiSourceFrameArrived +=
  FrameArrived;
```

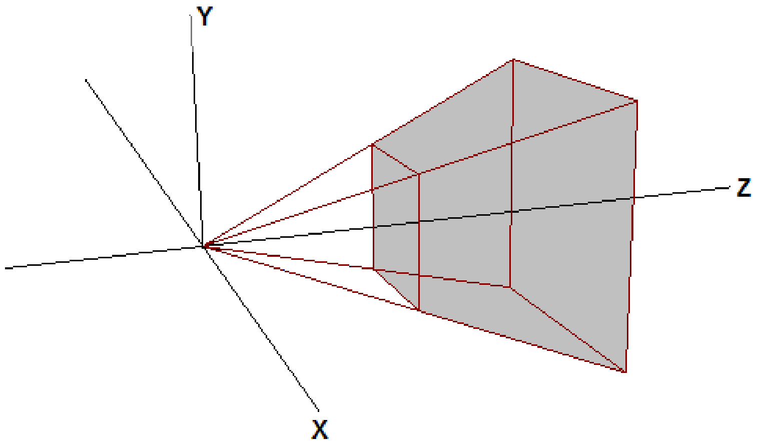
MultiSourceFrameReader – Event handler

```
private void FrameArrived(object sender, MultiSourceFrameArrivedEventArgs e)
{
  // Acquire the color frame
  using (ColorFrame frame = e.ColorFrameReference.AcquireFrame())
  {
    …
  }

  // Acquire the depth frame
  using (DepthFrame frame = e.DepthFrameReference.AcquireFrame())
  {
    …
  }

  …
}
```

Coordinate mapping



- Three coordinate systems

Name	Applies to	Dimensions	Units	Range	Origin
ColorSpacePoint	Color	2	pixels	1920x1080	Top left corner
DepthSpacePoint	Depth, Infrared, Body index	2	pixels	512x424	Top left corner
CameraSpacePoint	Body	3	meters	–	Infrared/depth camera

- Coordinate mapper provides conversions between each system
- Convert single or multiple points
- Many (but not all) methods require actual depth data

Coordinate mapping – Joint overlay

- Map the joints of a body to the color frame (e.g., to overlay joints on color image)
- We have multiple CameraSpacePoints, need ColorSpacePoints

```
int count = body.Joints.Count;
JointType[] jointTypes = body.Joints.Keys.ToArray();
CameraSpacePoint[] cameraPoints = body.Joints.Values.ToArray();
ColorSpacePoint[] colorPoints = new ColorSpacePoint[count];

this._sensor.CoordinateMapper.MapCameraPointsToColorSpace(
  cameraPoints, colorPoints);

for (int i = 0; i < count; ++i)
{
    DrawJoint(
      colorBitmap, colorPoints[i].X, colorPoint[i].Y, jointTypes[i]);
}
```

Coordinate mapping – Point cloud

- We have a frame of depth data, need CameraSpacePoints

```
int count = depthData.Length;
CameraSpacePoint[] pointCloud = new CameraSpacePoint[count];
this._sensor.CoordinateMapper.MapDepthFrameToCameraSpace(
  depthData, pointCloud);
```

Coordinate mapping – Color point cloud

- We have a point cloud of CameraSpacePoints, need corresponding color values
- PointWithColor is defined by the application: a struct containing a CameraSpacePoint and a Color
- NOTE: Very slow as written; “unsafe” code would yield much better performance

```
ColorSpacePoint[] colorPoints = new ColorSpacePoint[count];
this._sensor.CoordinateMapper.MapCameraPointsToColorSpace(
  pointCloud, colorPoints);

PointWithColor[] colorPointCloud = new PointWithColor[count];
for (int i = 0; i < count; ++i)
{
  colorPointCloud[i] = new PointWithColor
  {
    Position = pointCloud[i],
    Color = GetColor(colorData, colorFrameDesc, colorPoints[i])
  };
}
```

CoordinateMapping – GetColor

```
private Color GetColor(
  byte[] colorData, FrameDescription frameDesc, ColorSpacePoint colorPoint)
{
  if (colorPoint.X < 0 || colorPoint.X >= frameDesc.Width ||
      colorPoint.Y < 0 || colorPoint.Y >= frameDesc.Height)
  {
    return Colors.Transparent;
  }
  int index = ((colorPoint.Y * frameDesc.Width) + colorPoint.X) * frameDesc.BytesPerPixel;
  return new Color
    {
      B = colorData[index],
      G = colorData[index + 1],
      R = colorData[index + 2],
      A = colorData[index + 3]
    };
}
```

Porting guidance

High-level changes

- Applications that use one sensor
  - May no longer need sensor chooser component
  - KinectSensor.Default “just works”
  - Can use KinectSensor.IsAvailable, if necessary
- Many fewer flags/options (near/far, standing/seated, resolution)
- Stream model → Source/reader model
- AllFramesReady → MultiSourceFrameReader

All applications – Data changes

- Body index has its own source (separate from depth)
- Color format conversion at time of frame processing
- 6 bodies fully tracked: no need to choose
- More body joints (neck, thumbs, hand tips)
- *Some joints at different positions than before*
  (more anatomically correct, especially hip positions)
- Some joints renamed

New and renamed joints

- Code ported from V1 can safely ignore the new joints, and just use values 0 thru 19 of the JointType enum
- Most joints are the same in V1 and V2, except:

JointType Value	V1	V2
0	HipCenter	SpineBase
1	Spine	SpineMid
2	ShoulderCenter	Neck
20	–	SpineShoulder
21	–	HandTipLeft
22	–	ThumbLeft
23	–	HandTipRight
24	–	ThumbRight

COM applications

- New API model is much closer to .NET model
  - Significantly different API “shape” from COM V1 API
  - More interfaces, instead of monolithic INuiSensor interface
- Waitable event handles are provided by the API (not the application)

Microsoft

# KINECT
for Windows