Programming the Microsoft Kinect in Pascal

Michaël Van Canneyt

December 26, 2013

Abstract

The Kinect is a device created by Microsoft to enable NUI (Natural User Interface) for X-Box and Windows. It tracks the movement of the human body, and provides a stereoscopic image of whatever is located in front of the camera. The C++ API for this device can be used in Pascal.

1 Introduction

Some years ago, Microsoft introduced the Kinect for its game console XBox: a small camera-like device that sits on top of the TV, and registers players and their movements. The Kinect does for complete human bodies what the Leap Motion (introduced in an earlier article) does for hands: it can track the position and movement of the human body (called skeleton tracking), and provides a stereoscopic image (depth map) of whatever is located in front of the camera. In addition, the Kinect can also be used as a microphone (it accepts voice commands) and simply as a webcam. A picture of the device can be seen in figure 1 on page 1.

Since 2011, the Kinect SDK is also available for Windows PCs, and a C# (or .NET) and C++ API is available. The C# interface is more elaborate than the C++ interface, but the C++ interface is usable in all programming languages, including, as it turns out, Object Pascal.

The SDK can be downloaded for free from the MSDN developer website, and version 1.7 was used for this article (a new version is scheduled for Q1 2014). It contains some libraries (both for .NET and native development), which need to be distributed with an application that wants to connect to the Kinect. In particular, the kinect10.dll must be distributed.

The Microsoft Kinect C++ SDK provides roughly the same functionality as the open-source OpenNI and OpenCV libraries, but has a much more simplified API than the latter libraries.

Figure 1: The kinect for XBox 360
This article shows how to use the API for a simple skeleton-tracking application: discussing the complete SDK is beyond the scope of a single article.

2 Pascal Headers - The kinect API

The C++ SDK headers have been translated to Pascal. A version for Delphi is available on Google code:

http://code.google.com/p/kinect-sdk-delphi/

The Delphi units are called NuiAPI, NuiSensor, NuiImageCamera, SuiSensor and NuiSkeleton. Free Pascal ships a single unit libkinect10 that combines all definitions of the Delphi units. The names, structures and interfaces should be identical.

Both sets of units load the library dynamically. Loading and unloading the library must be done through the following functions:

Function LoadNuiLibrary(Const Filename : string = LibKinect) : Integer;
Procedure UnloadNuiLibrary;

When loading the library, the filename is optional and when none is specified, the default kinect10.dll is used. The free Pascal version of the units uses a reference counting mechanism, which means that UnloadNuiLibrary must be called as much as LoadNuiLibrary was called.

The kinect library does not need to be initialized: once it is loaded, it is ready for use. The library exposes a few global functions, and some interfaces.

One of these interfaces is INuiSensor, representing the Kinect camera. It has the following signature: (for brevity, the arguments of the methods have been omitted)

INuiSensor = interface(IUnknown)
  ['{1f5e088c-a8c7-41d3-9957-209677a13e85}']
Function NuiInitialize(dwFlags : DWORD) : HRESULT;
Procedure NuiShutdown;
Function NuiSetFrameEndEvent() : HRESULT;
Function NuiImageStreamOpen() : HRESULT;
Function NuiImageStreamSetImageFrameFlags() : HRESULT;
Function NuiImageStreamGetImageFrameFlags() : HRESULT;
Function NuiImageStreamGetNextFrame() : HRESULT;
Function NuiImageStreamReleaseFrame() : HRESULT;
Function NuiImageGetColorPixelCoordinatesFromDepthPixel() : HRESULT;
Function NuiImageGetColorPixelCoordinatesFromDepthPixelAtResolution() : HRESULT;
Function NuiImageGetColorPixelCoordinateFrameFromDepthPixelFrameAtResolution() : HRESULT;
Function NuiCameraElevationSetAngle() : HRESULT;
Function NuiCameraElevationGetAngle() : HRESULT;
Function NuiSkeletonTrackingEnable() : HRESULT;
Function NuiSkeletonTrackingDisable : HRESULT;
Function NuiSkeletonSetTrackedSkeletons() : HRESULT;
Function NuiSkeletonGetNextFrame() : HRESULT;
Function NuiTransformSmooth() : HRESULT;
Function NuiInstanceIndex : integer;
Not all of these methods will be discussed here, just the ones needed to track a skeleton and view the depth map.

The kinect library can be used with multiple kinect devices. The library therefore exposes some of the ISensor methods as global functions: if only a single kinect is connected to the computer, then these global functions can be used to control the kinect. Other than that no interface is used, the methods are the same, and the same procedures must be followed. Since the operating method is the same, in this article the more general approach using interfaces is used.

## 3 Detecting a kinect device

Once the kinect library is loaded, an interface to a kinect device must be retrieved. This can be done using 2 global functions:

Function NuiGetSensorCount(out Count : integer): HRESULT;
Function NuiCreateSensorByIndex(Index : integer; out ppNuiSensor : INuiSensor): HRESULT;

The NuiGetSensorCount function returns the number of connected Kinect devices. The NuiCreateSensorByIndex function then creates a INuiSensor interface for the Index-th device. Both functions return a HRESULT value: that means that the result of the function can be checked using the windows’ unit Failed function. The usable function result is always returned in out parameters.

Once a device has been detected, and an interface to the device was returned, the device must be initialized. When the device is no longer needed, the device can be shut down. These operations can be performed using the following methods of the INuiSensor interface:

Function NuiInitialize(dwFlags : DWORD) : HRESULT;
Procedure NuiShutdown;

When initializing the device, the device needs to be told what kind of processing it should do: calculate depth image, track player skeletons. Since each step in processing takes CPU time, it is important not to request processing that will not be used anyway. This is specified in the dwFlags option to the NuiInitialize function. The flags are an OR-ed combination of the following constants:

**NUI_INITIALIZE_FLAG_USES_AUDIO** Request audio data.

**NUI_INITIALIZE_FLAG_USES_COLOR** Request color data.

**NUI_INITIALIZE_FLAG_USES_DEPTH** Request depth data.

**NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX** Request depth data with a player index.

**NUI_INITIALIZE_FLAG_USES_SKELETON** Request skeleton tracking.
For the purpose of this article, only the last 2 will be used. The difference between the
`NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX` and
`NUI_INITIALIZE_FLAG_USES_DEPTH` is that the former encodes a player index in
the depth map: the depths are returned as word-sized values, and the 3 last bits of the word
are used to encode a player index (meaning that at most 7 players can be used).

4 Reading data from the device

The Kinect API provides several data streams: video, audio, depth map, skeleton data. The
API uses event handles to report the presence of data in one of these streams. That means
that the calling application needs to set up several event handles, one for each kind of data
stream it wishes to receive. These event handles must then be passed on to the data stream
initialization functions.

For the demo application, 2 streams will be examined: the depth map and the skeleton
tracking data. Both streams are provided through memory blocks that must be requested
through some methods of the `INuiSensor` interface.

The skeleton tracking stream is initialized (or stopped) through the following functions:

Function `NuiSkeletonTrackingEnable(hNextFrameEvent : THandle;` 
dwFlags : DWORD ) : HRESULT;
Function `NuiSkeletonTrackingDisable : HRESULT;`

The first parameter to `NuiSkeletonTrackingEnable` is the handle used to report the
presence of a new skeleton frame. The second parameter determines how the tracking data
is calculated and returned:

`NUI_SKELETON_TRACKING_FLAG_SUPPRESS_NO_FRAME_DATA` When set,
the `NuiSkeletonGetNextFrame` method will not return a `E_NUI_FRAME_NO_DATA`
error when no data is present, instead the call will block until data is present or the
timeout is reached.

`NUI_SKELETON_TRACKING_FLAG_TITLE_SETS_TRACKED_SKELETONS` When
set, the detected players are not really tracked. The `NuiSkeletonSetTrackedSkeletons`
must be used to select the players that should be fully tracked.

`NUI_SKELETON_TRACKING_FLAG_ENABLE_SEATED_SUPPORT` Enable seated
skeleton tracking. This means that the 10 lower-body joints of each skeleton are
not tracked, resulting in less calculations (the default is to track the whole body, 21
joints). When tracking a person seated in front of a computer, this option can be used
to reduce calculation time.

The depth image stream, as well as other image streams, are initialized through the follow-
ing functions:

Function `NuiImageStreamOpen(`
  eImageType : NUI_IMAGE_TYPE;
  eResolution : NUI_IMAGE_RESOLUTION;
  dwImageFrameFlags : DWORD;
  dwFrameLimit : DWORD;
  hNextFrameEvent : THandle;
  out phStreamHandle : THandle) : HRESULT;
Function `NuiImageStreamSetImageFrameFlags(`
  hStream : THandle;
  dwImageFrameFlags : DWORD) : HRESULT;
The NuiImageStreamOpen function opens an image stream. Which images the stream returns is specified through the eImageType parameter, which can have one of the following values:

- **NUI_IMAGE_TYPE_COLOR** a color image.
- **NUI_IMAGE_TYPE_COLOR_INFRARED** an infrared image
- **NUI_IMAGE_TYPE_COLOR_RAW_BAYER** a Raw Bayer color image (RGB)
- **NUI_IMAGE_TYPE_COLOR_RAW_YUV** a YUV color image; no conversion to RGB32.
- **NUI_IMAGE_TYPE_COLOR_YUV** a YUV color image, converted to RGB32
- **NUI_IMAGE_TYPE_DEPTH** a depth image.
- **NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX** a depth image with player index encoded in the map.

Various streams can be opened to capture data from the same device, but the capture of depth images must be enabled when initializing the device. The resolution of the image can be specified in the eResolution parameter, which can have one of the values

- NUI_IMAGE_RESOLUTION_80x60
- NUI_IMAGE_RESOLUTION_320x240
- NUI_IMAGE_RESOLUTION_640x480
- NUI_IMAGE_RESOLUTION_1280x960

The dwImageFrameFlags parameter can be used to specify some flags when capturing images, it accepts the same values as used in the NuiImageStreamSetImageFrameFlags function. The hNextFrameEvent parameter is the handle of the event that must be triggered when a new frame is ready.

Finally, the phStreamHandle is the handle of the image stream that must be used in the NuiImageStreamGetNextFrame calls to read the image.

The NuiImageStreamSetImageFrameFlags method can be used to modify the flags passed in the dwImageFrameFlags parameter to NuiImageStreamOpen:

- **NUI_IMAGE_STREAM_FLAG_DISTINCT_OVERFLOW_DEPTH_VALUES** Is undocumented.
- **NUI_IMAGE_STREAM_FLAG_ENABLE_NEAR_MODE** Enable near mode. (enable depth detection close to the camera)
- **NUI_IMAGE_STREAM_FLAG_SUPPRESS_NO_FRAME_DATA** When set, the NuiImageStreamGetNextFrame method will not return a E_NUI_FRAME_NO_DATA error when no data is present, instead the call will block until data is present or the timeout is reached.
- **NUI_IMAGE_STREAM_FLAG_TOO_FAR_IS_NONZERO** Is undocumented.

After the image stream and skeleton stream have been set up, frames can be read by watching the event handles. In the example program later on, this will be done in a separate thread.

### 5 Interpreting skeleton frame data

When a skeleton frame is ready, it can be fetched with the following method of INuiSensor:
Function NuiSkeletonGetNextFrame(
  dwMillisecondsToWait : DWORD;
  pSkeletonFrame : PNUI_SKELETON_FRAME) : HRESULT;

The first parameter is a timeout: if no frame is ready within the specified time, the call
returns with an error condition. When an event handle is used to signal the completion of a
frame, then the call should return at once.

The second parameter is a pointer to a NUI_SKELETON_FRAME structure. On return, it
points to a record that describes the tracked skeletons. It is described as follows

NUI_SKELETON_FRAME = record
  liTimeStamp: int64;
  dwFrameNumber,
  dwFlags: DWORD;
  vFloorClipPlane,
  vNormalToGravity: Vector4;
  SkeletonData : array[0..5] of NUI_SKELETON_DATA;
end;

The interesting data is the last structure, an array of 6 NUI_SKELETON_DATA records.
The limit of 6 skeletons is hardcoded: the kinect tracks at most 6 players (A constant exists
which describes this limit: NUI_SKELETON_COUNT). The 6 elements of the array are
always present, even if less skeletons have actually been detected.

Each skeleton is described by the following record:

NUI_SKELETON_DATA = record
  eTrackingState: NUI_SKELETON_TRACKING_STATE;
  dwTrackingID,
  dwEnrollmentIndex,
  dwUserIndex: DWORD;
  Position: Vector4;
  SkeletonPositions: array[0..19] of Vector4;
  eSkeletonPositionTrackingState: array[0..19] of
    NUI_SKELETON_POSITION_TRACKING_STATE;
  dwQualityFlags: DWORD;
end;

The eTrackingState field describes whether the record actually describes a skeleton.
It can have one of the following values:

NUI_SKELETON_NOT_TRACKED The record does not describe a tracked skeleton.

NUI_SKELETON_POSITION_ONLY The record describes a skeleton whose position
is tracked.

NUI_SKELETON_TRACKED record describes a fully tracked skeleton.

To determine the skeletons, the eTrackingState field of each record in the SkeletonData
array of NUI_SKELETON_FRAME must be checked. If it contains NUI_SKELETON_TRACKED,
it is a usable record.

For each skeleton, 20 joints are tracked. These joints are described in the SkeletonPositions
and eSkeletonPositionTrackingState arrays. For each of the 20 joints, a con-
tant is defined, for example: NUI_SKELETON_POSITION_HEAD,
NUI_SKELETON_POSITION_HAND_LEFT,
NUI_SKELETON_POSITION_HAND_RIGHT. Each of these constants is an index in the SkeletonPositions and eSkeletonPositionTrackingState arrays.

The eSkeletonPositionTrackingState array determines which of the SkeletonPositions elements contains a valid position vector. An element in the array can have one of the following values:

NUI_SKELETON_POSITION_NOT_TRACKED The array element does not contain valid data.

NUI_SKELETON_POSITION_INFERRED The position is calculated from other data.

NUI_SKELETON_POSITION_TRACKED The position is tracked.

The last 2 values mean that the element with the same array index in the SkeletonPositions array, contains a valid joint position.

The skeleton tracking mechanism may result in ‘jittery’ data. The results are vectors, and the positions will appear to have some random Brownian-like motion. The INuiSensor interface offers the NuiTransformSmooth function to deal with this:

Function NuiTransformSmooth(
    pSkeletonFrame : PNUI_SKELETON_FRAME;
    const pSmoothingParams :
        PNUI_TRANSFORM_SMOOTH_PARAMETERS) : HRESULT;

This function will attempt to reduce the randomness by applying a transformation on the received coordinates. The transformation is controlled by the following NUI_TRANSFORM_SMOOTH_PARAMETERS record:

NUI_TRANSFORM_SMOOTH_PARAMETERS = record
    fSmoothing,
    fCorrection,
    fPrediction,
    fJitterRadius,
    fMaxDeviationRadius : single;
end;

The exact meaning of these parameters can be found in the NUI API documentation on MSDN.

6 Interpreting depth image data

If a depth image is requested, the INuiSensor’s method NuiImageStreamGetNextFrame can be used to retrieve the actual depth image. It is declared as follows:

Function NuiImageStreamGetNextFrame(
    hStream : THandle;
    dwMillisecondsToWait : DWORD;
    pImageFrame : PNUI_IMAGE_FRAME) : HRESULT;

The hStream handle is an image stream handle created using the NuiImageStreamOpen function. Similar to the NuiSkeletonGetNextFrame function, the dwMillisecondsToWait specifies a timeout, in case the image is not yet ready.

On return, the location pointed to by pImageFrame will be filled with a NUI_IMAGE_FRAME record:
NUI_IMAGE_FRAME = record
  liTimeStamp: int64;
  dwFrameNumber: DWORD;
  eImageType: NUI_IMAGE_TYPE;
  eResolution: NUI_IMAGE_RESOLUTION;
  pFrameTexture: INuiFrameTexture;
  dwFrameFlags: DWORD;
  ViewArea: NUI_IMAGE_VIEW_AREA;
end;

The pFrameTexture field contains an INuiFrameTexture interface that can be used to examine the actual frame data:

INuiFrameTexture = interface(IUnknown)
  ['{13ea17f5-ff2e-4670-9ee5-1297a6e880d1}']
  Function BufferLen: integer;
  Function Pitch: integer;
  Function LockRect(Level: UINT;
    pLockedRect: PNUI_LOCKED_RECT;
    pRect: PRECT;
    Flags: DWORD ): HRESULT;
  Function GetLevelDesc(Level : UINT;
    out desc : NUI_SURFACE_DESC): HRESULT;
  Function UnlockRect(Level: UINT): HRESULT;
end;

For the depth image, the data comes in the form of an array of word-sized values. The byte size of the array is reported using BufferLen, the length of a single scan line can be retrieved with the Pitch method. The actual array can be retrieved with LockRect. Since the array is managed by the kinect driver, it is locked when it is retrieved. It must be unlocked using the UnlockRect call when it is no longer needed.

The data array is described by the following record

NUI_LOCKED_RECT = record
  Pitch : integer;
  size : integer;
  pBits : pointer;
end;

Where Pitch and Size correspond to the BufferLen and Pitch methods of the INuiFrameTexture interface. The pBits pointer points to the actual array.

Each element in the array is a Word value between NUI_IMAGE_DEPTH_MINIMUM_NEAR_MODE and NUI_IMAGE_DEPTH_MAXIMUM_NEAR_MODE if near mode is enabled. In normal mode, the minimum and maximum values are NUI_IMAGE_DEPTH_MINIMUM and NUI_IMAGE_DEPTH_MAXIMUM.

When NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX was used when creating the stream, the depth image’s word values are shifted, and the last 3 bits are used to encode a player index. (3 is the value of NUI_IMAGE_PLAYER_INDEX_SHIFT). If the last 3 bits are nonzero, then the pixel is considered part of a player’s body. The player index can be used for example to color the corresponding pixels in a player-specific color.
7 Putting everything together

After the long description of the Kinect (Natural User Interface) API, a small demonstration application will clarify things a bit. The sample application:

- Connects to the first found Kinect sensor.
- Requests and displays skeleton frames and a depth image stream.
- Uses events to get a notification when the next frames are ready.
- Displays the depth image with a specific color for all players, and superimposes on that, for the first detected player, shapes representing the hands and head.
- Allows to set/get the camera elevation angle.

The program is written in Lazarus, but it should work equally well in Delphi. It is a simple form, with 2 panels, some controls, and 3 shapes on it. The OnCreate event handler is used to initialize some variables and connect to the Kinect:

```delphi
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FESkeleton:=INVALID_HANDLE_VALUE;
  FEDepth:=INVALID_HANDLE_VALUE;
  FSDepth:=INVALID_HANDLE_VALUE;
  LoadNuiLibrary;
  TBAngle.Min:=NUI_CAMERA_ELEVATION_MINIMUM;
  TBAngle.Max:=NUI_CAMERA_ELEVATION_MAXIMUM;
  if not InitKinect then
    ShowMessage(‘Could not initialize kinect!’);
  For I:=1 to 6 do
    FPlayerColors[i]:=clWhite;
end;
```

The variables FESkeleton and FEDepth are the events used to receive notifications when the skeleton and depth frames are ready. The FSDepth variable will contain a handle for the depth image stream. The range of the camera’s elevation angle is determined by the NUI_CAMERA_ELEVATION_MINIMUM and NUI_CAMERA_ELEVATION_MAXIMUM constants (-27 and 27, respectively), these values are used to initialize a track bar control which can be used to set the angle of the Kinect’s camera.

Lastly, an array of colors is initialized to show the players on the depth map.

After loading the Kinect library, the InitKinect function is called to actually initialize everything:

```delphi
function TMainForm.InitKinect (EnableNear : Boolean = False): boolean;

var
  w,h : DWord;
  C,i : integer;
  NS : INuiSensor;
  E : Int64;

begin
  Result:=false;
```
FKinect := nil;
if Failed(NuiGetSensorCount(C)) then
  exit;
I:=0;
While (FKinect=Nil) and (i<C) do
  begin
    if Not Failed(NuiCreateSensorByIndex(i,NS)) then
      if (NS.NuiStatus=S_OK) then
        FKinect:=NS;
      Inc(I);
  end;
if not Assigned(FKinect) then
  exit;
if Failed(FKinect.NuiInitialize(NUIOptions)) then
  begin
    FKinect:=Nil;
    Exit;
  end;

This code is pretty straightforward, it requests the number of kinect devices, and connects to the first available one. If none was detected, it exits. The first available sensor is then initialized, the NUIOptions constant is defined as:

Const
NUIOptions =
  NUI_INITIALIZE_FLAG_USES_SKELETON or
  NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX;

After the kinect was initialized, an event handle is created, and used to enable skeleton tracking:

FESkeleton:=CreateEvent(nil,True,False,nil);
FKinect.NuiSkeletonTrackingEnable(FESkeleton,SkeletonOptions);

SkeletonOptions is a constant requesting seated support and near range. The next thing to do is request a depth image, again using an event handle to get notifications:

FEDepth:=CreateEvent(nil, true, false, nil);
if Failed(FKinect.NuiImageStreamOpen(ImageOptions,
  ImageResolution, 0, 2, FEDepth, FSDepth)) then
  Exit;
if EnableNear then
  if Failed(FKinect.NuiImageStreamSetImageFrameFlags(FSDepth, ImageStreamOptions))
    Exit;

If all went well, a thread can be set up to check for events on the FESkeleton and FEDepth

FTEvents:=TEventDispatcherThread.CreateDispatcher(Handle,
  FESkeleton,
  FEDepth);

Lastly, the size of the depth image is used to create a bitmap (which will be used to draw the depth image) and set the width and height of the form: FEDepth.
NuiImageResolutionToSize(ImageResolution, w, h);
ClientWidth:=w;
ClientHeight:=h;
FBDepth:=TBitmap.Create;
FBDepth.Width:=w;
FBDepth.Height:=h;
Result:=true;
if Not Failed (FKinect.NuiCameraElevationGetAngle(@A)) then
  TBAngle.Position:=A;
end;

The last statements retrieve the elevation angle of the kinect’s camera, and initialize a track-bar (TBAngle) with the current position of the camera.

The TEventDispatcherThread is a thread descendant (in the EventDispatcherThread unit) which simply loops and waits for kinect events. When a kinect event is detected, a windows WM_USER message is sent to the main form.

This is one way of handling the events, another way would be to use the OnIdle event of the application, or a timer, to check for new events. Instead of sending a message, it is also possible to use the Synchronize or Queue methods to let the main thread respond to the arrival of new data.

The thread's Execute method looks very simple:

procedure TEventDispatcherThread.Execute;
begin
  if (FHWnd=INVALID_HANDLE_VALUE) or
     (FESkeleton=INVALID_HANDLE_VALUE) then
    exit;
  While not terminated do
    begin
      if (WaitForSingleObject(FESkeleton,50)=WAIT_OBJECT_0) then
        begin
          SendMessage(FHWnd,WM_USER,MsgSkeleton,0);
          ResetEvent(FESkeleton);
        end;
      if (WaitForSingleObject(FEDepth,50)=WAIT_OBJECT_0) then
        begin
          SendMessage(FHWnd,WM_USER,MsgDepth,0);
          ResetEvent(FEDepth);
        end;
    end;
end;

If an event is received on either of the 2 handles, a WM_USER message is sent to the main form, which will then take appropriate action. The message parameters are defined as constants. Note that the event is reset after the message is sent.

Reacting on the messages is done by implementing a message handler method in the form for the WM_USER message:

procedure TMainForm.eventDispatcher(var msg: TMessage);
begin
  if (msg.WParam=msgSkeleton) then
    OnNewSkeletonFrame
  else if (msg.WParam=msgDepth) then

11
OnNewDepthFrame;
DoTick(msg WPARAM=msgDepth);
end;

The message handler method simply examines the message parameter and calls the appropriate method to deal with the message. After that it executes a tick, which collects and displays some statistics.

The actual work is done in OnNewSkeletonFrame and OnNewDepthFrame. The first one is responsible for drawing the head and hand joints in the skeleton frame.

It examines the received data and positions 3 shapes on the form:

```pascal
procedure TForm1.OnNewSkeletonFrame;
var
  i : integer;
  fr : NUI_SKELETON_FRAME;
  PSD : PNUI_SKELETON_DATA;
  tsp : NUI_TRANSFORM_SMOOTH_PARAMETERS;
begin
  FillChar(fr,sizeof(NUI_SKELETON_FRAME),0);
  if Failed(FKinect.NuiSkeletonGetNextFrame(0,@fr)) then
    Exit;
  PSD:=Nil;
  i:=0;
  While (PSD=Nil) and (i<NUI_SKELETON_COUNT) do
    begin
      if (fr.SkeletonData[i].eTrackingState=NUI_SKELETON_TRACKED) then
        PSD:=@fr.SkeletonData[i];
      Inc(i);
    end;
  if Not Assigned(PSD) then
    Exit;

  This code fetches the next NUI_SKELETON_FRAME structure from the kinect, and initializes a pointer to the first skeleton (PSD). The following step is 'smoothing out' the received coordinates:

  With tsp do
    begin
      fCorrection:=0.3;
      fJitterRadius:=1.0;
      fMaxDeviationRadius:=0.5;
      fPrediction:=0.4;
      fSmoothing:=0.7;
    end;
  if Failed(FKinect.NuiTransformSmooth(@fr,@tsp)) then
    Exit;

  And finally, the 3 shapes are positioned:

  ShowJoint(SHead,PSD,NUI_SKELETON_POSITION_HEAD);```
ShowJoint(SLeft,PSD,NUI_SKELETON_POSITION_HAND_LEFT);
ShowJoint(SRight,PSD,NUI_SKELETON_POSITION_HAND_RIGHT);
end;

The `ShowJoint` will check if the requested joint position (the third parameter) was tracked, and if so, position the shape so it is centered on this position.

To position a shape on the depth bitmap, the skeleton coordinates must be transformed to a X,Y position on the depth image. This can be done with the aid of the `NuiTransformSkeletonToDepthImage` function:

```
Procedure NuiTransformSkeletonToDepthImage(
    vPoint : TVector4;
    out fDepthX : single;
    out fDepthY : single;
    eResolution : NUI_IMAGE_RESOLUTION);
```

It receives a position vector, and a resolution. It returns an X,Y coordinate which is a coordinate on a depth bitmap corresponding to the given resolution.

All this is used in the `ShowJoint` function:

```
Procedure TMainForm.ShowJoint(S : TShape; PSD : PNUI_SKELETON_DATA; HI : Integer);
Var
    x, y : single;
begin
    S.Visible:=PSD^.eSkeletonPositionTrackingState[HI]=
        NUI_SKELETON_POSITION_TRACKED;
    if S.Visible then
        begin
            NuiTransformSkeletonToDepthImage(PSD^.SkeletonPositions[HI],
                x,y,
                NUI_IMAGE_RESOLUTION_640x480);
            S.Left:=Round(x)-(S.Width div 2);
            S.Top:=Round(y)-(S.Height div 2);
        end;
end;
```

Finally, the depth image must be rendered: for this, the depth image needs to be interpreted and transferred to a bitmap, and then the bitmap is drawn on a panel:

```
procedure TMainForm.OnNewDepthFrame;
Const
    DS = NUI_IMAGE_PLAYER_INDEX_SHIFT;
    MINS = NUI_IMAGE_DEPTH_MINIMUM_NEAR_MODE shr DS;
    MAXS = NUI_IMAGE_DEPTH_MAXIMUM_NEAR_MODE shr DS;
var
    IFDepth : NUI_IMAGE_FRAME;
    FT : INuiFrameTexture;
    lck : NUI_LOCKED_RECT;
    depth : pword;
```
CD : Word;
p : byte;
x, y : integer;
w, h, GS : cardinal;
C : TCanvas;

begin

if (FSDepth=INVALID_HANDLE_VALUE) then
  Exit;
if Failed(FKinect.NuiImageStreamGetNextFrame(FSDepth,0,@IFDepth)) then
  Exit;
NuiImageResolutionToSize(IFDepth.eResolution, w, h);

The above code retrieves the image from the kinect and calculates a width and height with it.

The next step is to retrieve the image data from the INuiFrameTexture interface:

try
  FT:=IFDepth.pFrameTexture;
  if not assigned(FT) then
    Exit;
  if Failed(FT.LockRect(0,@lck,nil,0)) then
    Exit;
  try
    if lck.Pitch<>>(2*w) then
      Exit;
    depth:=lck.pBits;
  except
    exit;
  end;

The following steps transfer are a loop over the depth data, transferring it as a grayscale to the bitmap. The depths that have a player index in them are transferred to the bitmap using the player’s color. If there is no player index, the depth value is transformed to a grayscale value ranging from 0 to 255. Note that the bitmap canvas is locked for better performance:

C:=FBDepth.Canvas;
C.Lock;
for y:=0 to h-1 do
  for x:=0 to w-1 do
    begin
      CD:=Depth^;
      P:=(CD and NUI_IMAGE_PLAYER_INDEX_MASK);
      if (P<>0) then
        C.Pixels[X,Y]:=FPlayerColors[p]
      else if (CD>=NUI_IMAGE_DEPTH_MINIMUM_NEAR_MODE) and
        (CD<=NUI_IMAGE_DEPTH_MAXIMUM_NEAR_MODE) then
        begin
          GS:=Round(((CD shr ds) - MINS) / MAXS * 255);
          GS:=GS and $FF;
          GS:=GS or (GS shl 8) or (GS shl 16);
          C.Pixels[X,Y]:=GS;
        end
      else
        C.Pixels[X,Y]:=clBlack;
      Inc(depth);
    end

Lastly, the retrieved depth image data is released, and the bitmap is drawn on the panel:

```pascal
finally
    FT.UnlockRect(0);
end;
finally
    FKinect.NuiImageStreamReleaseFrame(FSDepth, @IFDepth);
end;
PImage.Canvas.Draw(0,0,FBDepth);
For X:=0 to PImage.ControlCount-1 do
    if PImage.Controls[X] is TShape then
        PImage.Controls[X].Repaint;
end;
```

Finally, to make sure the shapes representing the head and hands are properly shown, they are repainted.

The form contains some logic to display a tick and average frames per second count, and to set the colors of the players. This logic is not relevant to the operation of the kinect.

However, the routine to set the camera’s elevation angle needs some explanation: Setting the elevation angle of the camera takes some time: it is a mechanical operation, involving a small motor inside the kinect. Setting a new position while the previous position was not yet established, will result in an error. It is therefore important not to send commands too often or too much. The following code attempts to do that:

```pascal
procedure TMainForm.TBAngleChange(Sender: TObject);
Var
    A : Longint;
begin
    If TBAngle.Position=FLastPosition then
        Exit;
    if Not Failed (FKinect.NuiCameraElevationGetAngle(@A)) then
        begin
            if (A<>-TBAngle.Position) then
                A:=-TBAngle.Position;
            begin
                if Failed (FKinect.NuiCameraElevationSetAngle(A)) then
                    ShowMessage(Format(SErrSetAngle,[A]));
                FLastPosition:=-A;
            end;
        end;
    else
        ShowMessage(SErrGetAngle);
end;
```

Note that the trackbar position is reversed; it is positioned vertically, with the minimum value (-27) at the top, and the maximum value (27) at the bottom of the trackbar.

Everything put together, the running program results in a figure like figure 2 on page 16.
8 Conclusion

The Kinect is a device which is one way of implementing a Natural User Interface: use the human body to control the computer. While it is originally aimed at gaming, there may be specialized uses for this device outside the gaming industry. For a more fine-grained control of the computer, the resolution of the Kinect’s skeleton detection is not big enough: it cannot detect fingers of hands this gap may be better filled by the Leap Motion device. Both devices are available to Object Pascal programmers, and there are certainly Object Pascal game programmers that will consider the ability to use the Kinect a nice addition to their range of available tools in Pascal.