A Novel Approach for Constructing Emulator for Microsoft Kinect XBOX 360 Sensor in the .NET Platform

Mohammad Raihanul Islam, Sazzadur Rahaman, Rakibul Hasan, Ridwan Rashid Noel, Asif Salekin,
Hasan Shahid Ferdous, Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology, Bangladesh
{raihan_2108, sazzad14, rakib_cse_062, noel_hbk_008, asalekin, webtonmoy}@yahoo.com

***Abstract*—The Microsoft Kinect sensor has brought a new era of Natural User Interface (NUI) based gaming and the associated SDK has provided access to its powerful sensors, which can be utilized in many ways, especially in research purposes. We have already seen its use in robotics, developing assistive technologies, and augmented reality, aside from gaming. Thousands of people around the world are playing with its built-in multimodal sensors, but still a complete emulator for the Kinect sensor device is lacking, thus requiring a physical device to do any experiments with it. In this work, we have come forward with a novel design of an emulator for the Kinect sensor and its implementation in the .NET platform using the Microsoft Kinect SDK. We have demonstrated the applicability of our system through detailed software design, code descriptions to incorporate this emulator in user's own code, and video demonstration of our proposed system.**

***Keywords*-Kinect sensor; emulator; .NET platform.**

I. Introduction

Kinect is a motion sensing input device by Microsoft for the Xbox 360 video game console and Windows PCs. It allows the user to interact with the XBOX gaming machine without using a game controller, enabling the *you are the controller* paradigm. The concept of Kinect is base on Natural User Interface (NUI) - controlling the devices through gestures and voice commands. After its release on November, 2010, the Kinect holds the Guinness World Record of being the "fastest selling consumer electronics device", as a total of 8 million units are sold in its first 60 days. 18 million units of the Kinect sensor had been shipped as of January 2012 [1].

The Microsoft Kinect device has multimodal sensors - an RGB camera, an infrared depth sensor, and a microphone array to capture the surrounding environment. It runs with proprietary software which provides full-body 3D motion capture, facial recognition, and voice recognition capabilities. The applications using Kinect sensor access the sensor data through NUI API library interface calls (Fig. 1). To facilitate the application development and enhance research scopes using Kinect, Microsoft has released Kinect Software Development Kit (SDK) for Windows 7 on June 2011. Another open source SDK named *Open Kinect* was there since November 2010. So lots of researchers and enthusiastic students have worked with the Kinect sensor device exploiting its use it many areas.

One major challenge in working with a sensor device like Kinect is that we require the hardware in every stage of the software development. Using Kinect, we are working with raw sensor data, so many approaches require fine-tuning of the code with empirical data values. We can say about one practical challenge we faced when developing a *PC mouse control using Kinect* software. We required adjusting the software parameters painstakingly to make it work smoothly. One of the major challenges in that project is that we always required to work with the Kinect sensor to debug or tune our code. The severity of the problem lies in the fact that we had to do the same gestures again and again to test our code and tune it. Again, we had only one device for our five member team and the device costed about 100 USD, thus making the problem more complex. We practically felt the necessity of a emulator device for Kinect during that project development.

In computing, an emulator is a hardware or software or both that can imitate the functions of a device or system in a different system or device. In that process the second system behave closely like the original system. We can use emulators in many cases of software development. For example, there is Android phone emulators to enable programmers to develop software for android phones without requiring an android phone to develop and test the application. There are other emulators, like 8086 emulator, GPS emulator, etc. But a complete emulator for the Kinect sensor it not available till date. In this work, we are going to take the challenge and provide one.

There has been some works on developing a Kinect emulator before ours. Among them the most notable is the *Fakenect* by Brandyn White [2] with his libfreenect API. Fakenect was originally developed in python and it works only in Linux and Mac OSX environment, so users accustomed to Windows cannot use Fakenect directly. There have been some modifications by different user groups for Fakenect that could be supported in Windows too. It has some unresolved issues about implementation and synchronization. The original version of Fakenect does not include the implementation of audio features of Kinect. Also, it did not use any compression techniques in storing raw data, so the recorded streams took too much space in a very short

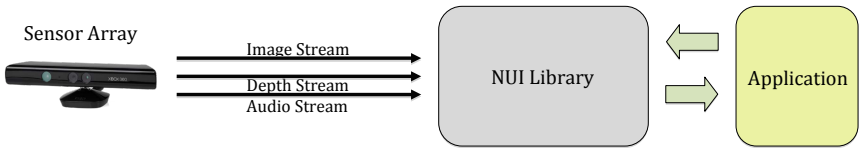

Figure 1. Interaction between the Kinect software and hardware [4].

time. We will discuss about these issues and ways to solve them in later sections of this paper.

In our work, we have focused to develop a Kinect emulator using the Microsoft Kinect SDK for the .NET platform so that people can integrate it and use seamlessly with their existing code. We have recorded all the raw data streams (including audio) from the device with necessary compressions to reduce the amount of data files and then redirected the recorded streams to make the system believe that it is using the Kinect device instead of recorded data from the storage drives. Our process is properly documented and the users require to change only a few lines in their code to switch between emulator mode and using real Kinect devices.

The rest of the paper is organized as follows. Section II familiarizes us with different sensors in the Kinect device and their properties. We present out approach in building a Kinect emulator briefly in Section III. In Section IV, we describe the background knowledge and terminologies required to understand our work. In Section V, we present our approach to store the skeleton data, while In Section VI, we present our contribution to store video and depth data. We show our technique to store audio data in Section VII. Finally we discuss about using our emulator in users' own code in the conclusion section.

II. Kinect Sensor Descriptions

We can see the logical interactions between the hardware and the software components in Microsoft Kinect in Fig. 1. Three types of data streams are available from the Kinect sensor: 1) Image Stream, 2) Depth Stream, and 3) Audio Stream. In this section, we summarize different properties of these streams (Table I).

A. Image Stream

The RGB video stream uses 8-bit VGA resolution (640 X 480 pixels). Color data is available in the following two formats:

- RGB color can be provided in 32-bit, linear X8R8G8B8-formatted color bitmaps, in the sRGB color space. An application must specify a color or color_YUV image while opening the stream.
- YUV color provides 16-bit, gamma-corrected linear UY VY-formatted color bitmaps, where the gamma correction in YUV space is equivalent to sRGB gamma in RGB space. Since YUV stream uses 16 bits per pixel, its memory requirement is smaller but YUV data is available only at 640X480 pixel and at 15 fps [4].

Both color formats are computed from the same camera data, so that the YUV data and RGB data represent the same image.

B. Depth Stream

The depth sensor has an infrared laser projector with a monochrome CMOS sensor, which is capable of capturing depth data in 3D under any ambient light conditions. The depth data stream provides frames in which each pixel represents the Cartesian distance, in millimeters, from the camera plane to the nearest object at that particular x and y coordinate in the depth sensor's field of view. The following resolutions in depth data streams are available: 1) 640X480 pixels, 2) 320X240 pixels, and 3) 80X60 pixels.

There is another type of data available form Kinect called Skeleton Data. Kinect can locate the twenty points of a player's body. The NUI Skeleton API of Kinect provides information about these points of a person standing in front of the Kinect sensor array. The data is provided to application code as a set of points, called skeleton positions, which compose a human skeleton [4].

C. Audio Stream

Kinect supports its audio features by implementing a microphones array consisting of 4 microphones arranged in a linear or L-shaped pattern. Implementing a set of microphones has some significant benefit over a single microphone, like capturing high quality audio, beam forming and source localization and speech recognition facilities.

III. Our Approach for Building the Kinect Emulator

Microsoft has released their Kinect software development kit (SDK) for Windows 7 on June 16, 2011 [4]. This SDK has allowed the developers to write Kinect applications in C++/ CLI, C#, or Visual Basic.NET. Developers around the world are trying to use the capabilities of Kinect in different fields. One of our previous works implementing Natural User Interface (NUI) based classroom using Kinect has been published in [5]. There are also some works based on Kinect such as touch free exploration of medical image data [3], tracking 3D position, orientation, and full articulation of a human hand [6].

Table I
SENSOR CAPABILITIES OF MICROSOFT KINECT DEVICE

Sensor item	**Playable range**
Color and depth stream	4 to 11.5 feet (1.2 to 3.5 meters)
Skeletal tracking	4 to 11.5 feet (1.2 to 3.5 meters)
Viewing angle	43° vertical by 57° horizontal field of view
Mechanized tilt range (vertical)	±28°
Frame rate (depth and color stream)	30 frames per second (FPS)
Resolution, depth stream	QVGA (320 x 240 pixel)
Resolution, color stream	VGA (640 x 480 pixel)
Audio format	16-kHz, 16-bit mono pulse code modulation (PCM)
Audio input characteristics	A four-microphone array with 24-bit analog-to-digital converter (ADC) and Kinect-resident signal processing such as acoustic echo cancellation and noise suppression

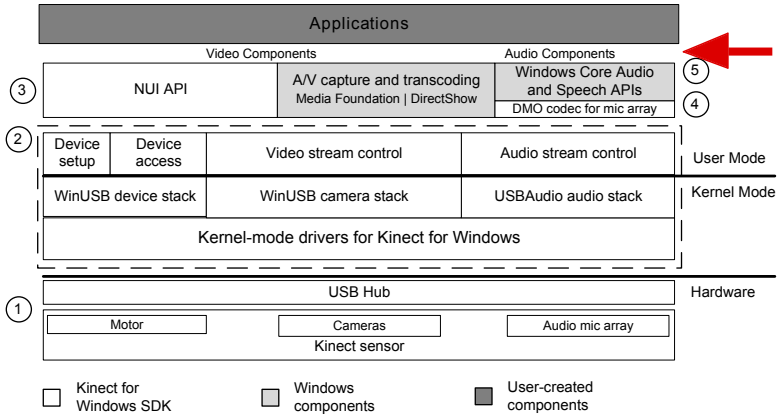

Figure 2. Kinect software architecture [4].

In normal operations, a Kinect device needs to be connected to the XBOX 360 or PC via USB port for the access and manipulation of the data streams provided by Kinect. Instead a Kinect emulator can be used, where a Kinect is not needed physically. An emulator takes the form of a hardware device and duplicates (or emulates) the functions of the device in a different second device, so that the emulated behavior closely resembles the behavior of the real device or system. Using an emulator does not need the presence of the original device. So, using a Kinect emulator, we can record data one time and use the recorded data when Kinect is not available to us. Reproducing results is easier in Kinect emulator and it does not constraint one to real-time performance. It allows one to record Kinect data and switch between live and recorded modes easily and also reduces the tedious job of giving the same human input several times during software development.

To construct an emulator we have to capture these streams of data and store it into proper format so that later, users can use these data as if they have got these from a Kinect device. We represent the complete software architecture of Kinect in Fig. 2. Our contribution has been implemented between the application and video and audio components shown by the big red arrow in Fig. 2.

IV. PRELIMINARIES

In this section, we describe the basic technologies we have used in our study. We describe our approach in data compression and decompression, key frame, etc.

A. *Data compression*

In computer science and information theory, data compression, source coding, or bit-rate reduction involves encoding information using fewer bits than the original representation. Compression can be either lossy or lossless. Lossy compression reduces bits by identifying marginally important information and removing it. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. Thus when Lossless compressed data is decompressed, it is possible to get back to the original data, but on the other hand this claim is not true for the lossy compression. We have used lossless data compression in our emulator to produce the exact same data that a real Kinect device could have offered.

B. *LZW Compression/Decompression:*

This is the most commonly used and the simplest type of compression and decompression algorithm. There are mainly

two types of implementations in LZW compression - static and dynamic compression.

1) Static Compression:: In static compression, we use a fixed number of bits to represent every character while compressing. The total number of bits we have at our disposal is 32, which gives a total of 2^{32}, that is, $4,294,967,295$ possible entries in the dictionary. But in almost all cases we do not get that much entries, so in most of the cases, we use a constant number of bits and commonly set the number of bits as 14. For example, in 14 bit format, 'A' can be represented by 00000001000001.

2) Dynamic Compression:: Dynamic compression changes the number of bits used to compress the data. It starts with 9 bits for each new value, and goes up until it reaches 32 or until the file ends. In this type of compression we can set the size of the dictionary we have. The dictionary begins with all the 256 ASCII codes. In dynamic compression there is only one leading 0 in front of each number. For example, in dynamic compression, 'A' can represented as 001000001.

C. Key frame:

In video compression, a keyframe, also known as an *Intra Frame*, is a frame in which a complete image is stored in the data stream. In video compression, only changes that occur from one frame to the next are stored in the data stream, in order to greatly reduce the amount of information that must be stored. This technique capitalizes on the fact that most video sources (such as a typical video stream from Kinect) have only small changes in the image from one frame to the next. Whenever a drastic change to the image occurs, such as when the background scene changes, a keyframe must be created. The entire image for the frame must be in the output when the visual difference between the two frames is so great that representing the new image incrementally from the previous frame would be more complex and would require even more bits than reproducing the whole image. Aside from the keyframes, we need to store the difference in color information from its preceding frame which, when compressed can result in very small data size. We have used this concept in our emulator to reduce the storage file size for raw video data. To increase efficiency, we generated the LZW dictionary for keyframes only and used it for other frames. It reduced our algorithmic complexity greatly and made our approach applicable in real-time.

D. Retrieving Data Streams from the Kinect Device

User applications can get the latest frame of image/depth/ skeletal data by calling a frame retrieval method and passing a buffer. If the latest frame of data is ready, it is copied into the buffer. If our code requests frames of data faster than new frames are available, we can choose whether to wait for the next frame or to return immediately and try again later. The NUI API never provides the same frame of data more than once. Applications can use either of the following two usage models:

1) Polling Model: When using the polling model, the application code opens the stream first. It then requests a frame and specifies how long to wait for the next frame of data (between 0 and an infinite number of milliseconds). The request method returns when a new frame of data is ready or when the wait time expires, whichever comes first. Specifying an infinite wait causes the call for frame data to block and to wait as long as necessary for the next frame.

When the request returns successfully, the new frame is ready for processing. If the time-out value is set to zero, the application code can poll for completion of a new frame while it performs other work on the same thread. For example, a native C++ application calls `NuiImageStreamOpen()` method to open a color or depth stream and omits the optional event. Managed code calls the `ImageStream.Open()` function. To poll for the color and depth frames, native C++ applications call `NuiImageStreamGetNextFrame()` and managed code calls `ImageStream.GetNextFrame()`.

2) Event Model: The event model supports the ability to integrate retrieval of a image/depth/skeleton frame into an application engine with more flexibility and more accuracy. In this model, C++ application code passes an event handle, for example, to `NuiImageStreamOpen()` method for image data. When a new frame of image data is ready, the event is signaled. Any waiting thread wakes and gets the frame of skeleton data by calling `NuiImageGetNextFrame()`. During this time, the event is reset by the NUI Image Camera API.

In the .NET model, Managed code uses the event mode by hooking a `Runtime.DepthFrameReady()` method for the depth stream or `Runtime.ImageFrameReady()` method for the image stream to an appropriate event handler. When a new frame of data is ready, the event is signaled and the handler runs and calls `ImageStream.GetNextFrame()` to get the frame.

V. Skeleton Data

Here in this section we present our approach on storing and retrieving the skeleton frame using our Kinect emulator. We get the position of each body joint of the person under consideration and then record it in the permanent storage devices, for example, the hard disk drive.

We used the event driven technique to retrieve the skeleton data and save it to file. When a skeleton frame is available and an event is triggered, the `void Nui_GotSkeletonAlert()` is invoked and an instance of `NUI_SKELETON_FRAME` is retrieved from `NuiSkeletonGetNextFrame()` function which contains the coordinate of the body points and then the whole frame is stored. We also record the time stamp difference from the previous skeleton frame.

When retrieving this data and redirecting them to our application instead of using the *real* Kinect device, we use

a thread to fetch data from that recorded file according to timestamp and invoke that particular event with event parameters passed manually in the same time gap as in the original data. In this way the user code remains almost transparent and only required to be changed in the Kinect initialization part, where instead of Kinect initializing, we are initializing our own thread for fetching data. The process is documented in [7].

VI. Video Data and Depth Data

In the scope of `void Nui_GotVideoAlert()` function, we retrieve the video data and persist it. But there are several issues we had to deal with while saving the video data. The Issues are listed below:

- As Kinect device outputs the video frame in the memory and the operation of writing a frame from the memory to a file is much slower than then frame generation rate of the Kinect device, we used a buffer to deal with the problem. Kinect writes data in the buffer and a thread reads data from the buffer and writes to the hard disk block by block.
- The size of the file where raw video frames are persisted grows so fast that in few minutes it crossed 4 or 5 GB in size. So we had to compress the frames. In that case we used LZW compression algorithm to compress the data as discussed earlier. We could not use established compression standards like .mp4, .flv, or .mpeg as they will compress the frames with their proprietary algorithms, can drop frames if necessary, includes their own metadata, and they are not lossless. We need to be very precise with the timestamp and metadata of the frames we store so that we can reproduce it exactly as it was recorded in the first place. So we used our own code for storing video data.
- Writing frames in file itself is a relatively slow operation and the compression process make it slower and it hugely influences the overall performance(frame rate per second).
- The Dictionary is created only when a keyframe comes and this made an viewable impact in performance but still we require to improve it further.
- So finally we decided not to record all the frames but to skip some of them and thus made it faster. The whole process is shown on Figure 3.

Similar as for the video data, we retrieve the depth data from the `Nui_GotDepthAlert()` function and persist it. When a depth frame event occurs, `Nui_GotDepthAlert()` retrieves the ready frame from the capture engine.

For both video and depth data, we recorded the received frames along with time stamps. So the retrieving thread, when using in our applications can invoke event calls with precise timing as discussed in the skeletal data section (Section V).

Table II
RIFF HEADER

Positions	Field Name	Size (Bytes)	Sample Value
1-4	chunkID	4	"RIFF"
5-8	ChunkSize	4	640022
9-12	Format	4	"WAVE"
13-16	subChunk1ID	4	"fmt"
17-20	subChunk1Size	4	16
21-22	AudioFormat	2	1
23-24	NumChannels	2	1
25-28	SampleRate	4	16000
29-32	ByteRate	4	32000
33-34	BlockAlign	2	2
35-36	BitsPerSample	2	16
37-40	subChunk2ID	4	"data"
41-44	subChunk2Size	4	640000
44	data	subChunk2Size	Audio Stream

VII. Audio Data

In this section, we describe our approach to store and retrieve the audio data from Kinect. As previously mentioned, Kinect audio features are supported by a microphone array. Generally the microphone array consists of 4 microphones. First we describe how use the Audio API to capture the audio stream and then describe the method we use to store source information.

A. Our Approach

Our goal is to capture the audio stream from the Kinect sensor's microphone array without losing any information. We have to store the audio stream and the information about beam forming and source localization with proper synchronization with the original stream as if it comes from the microphone array when someone use the stored audio data later. In our approach we write the audio data in ".wav" format because it is a standard audio format used in Windows and we have stored the source information in a separate file. In our approach we save the beam angle, source angle, and the level of confidence of source angle when there is a change in either angle with proper timestamps.

B. Capture the Audio Stream

First, we have to create an object to manage the Kinect's microphone array. Microsoft has provide a class in its NUI API named "`KinectAudioSource`" to manage the functionality for microphone array. We can use it to capture the audio stream and write the data stream into a ".wav" file. We made the thread priority highest for this program. To start the recoding we call the function `KinectAudioSource :: Start()` and we record the audio stream until the total recoding time passes. Here we record the audio at a sample rate of 16 KHz. The RIFF (Resource Interchange File Format) format was created by Microsoft and is used by many applications like Windows, Corel Draw,

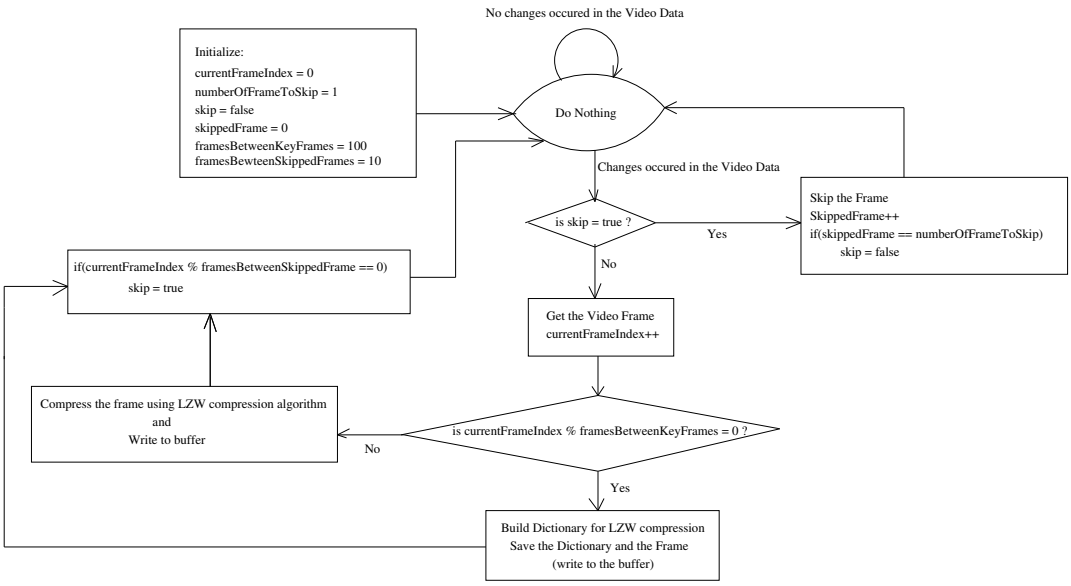

Figure 3. Capturing video data.

etc. We present the RIFF format and the values we used in its header in Table II.

C. Storing Audio Information

In this section we describe how we store the audio source information. We construct a class named `AudioData` for this purpose. The short description for `AudioData` is given below:

```
class AudioData{public :
double sourceAngle, beamAngle, confidenceSource;
_int64 ticks;};
```

Here the `sourceAngle`, `beamAngle` attributes are self explanatory. The `confidenceSource` is the confidence level of the source ranging from 0 to 1 as provided by the NUI API specifications. The last attribute `ticks` is the elapsed ticks after the previous entry of the changing of source or beam angle. A single tick represents one hundred nanoseconds or one ten-millionth of a second. There are 10,000 ticks in a millisecond as specified in the .NET documentation. The data type for tick is 64 bit integer for storing large value. Now we monitor the source direction. When there is change in the either angle we store in the file with the time elapsed after the last change. In this way we can store the complete information of the audio data. We can retrieve the audio information in the same way for video and skeleton data, as mentioned in earlier sections.

VIII. Conclusion

In this paper we have presented a novel approach for making a Kinect device emulator for Windows platform. Our system can record raw data streams from a real Kinect device and later everyone can use that offline data to experiment with their own application without having a Kinect device. This will greatly enhance group research and development activities as we no longer require a physical Kinect device for every member of the group. Users worldwide can download our emulator code with documentation to use it from [7]. A video demonstration showing the usability of our emulator can be found in [8]. We are now looking forward to improve our emulator further from the user feedbacks we receive.

References

[1] "http://venturebeat.com/2012/01/09/xbox-360-sur-passed-66m-sold-and-kinect-has-sold-18m-units/", Tak ahashi, Dean (January 9, 2012). "Xbox 360 surpasses 66M sold and Kinect passes 18M units". Retrieved June 13, 2012.

[2] "http://brandynwhite.com/ fakenect-openkinect-driver - simulator-experime", Last accessed : May 25, 2012.

[3] L. Gallo, A. P. Placitelli, and M. Ciampi. Controller free exploration of medical image data: Experiencing the Kinect. In *Proc. CBMS*, pages 1–6, 2011.

[4] *Programming Guide Kinect SDK*, Microsoft Research.

[5] R. R. Noel, A. Salekin, R. Islam, S. Rahaman, R. Hasan, and H. S. Ferdous. "A natural user interface classroom based on Kinect". In IEEE Learning Techn- ology Newsletter, volume 13, October 2011.

[6] I. Oikonomidis, N. Kyriazis, and A. A. Argyros. Effici- ent model-based 3d tracking of hand articulations usi- ng Kinect. in *Proc. BMVC, 2011.*

[7] "https://github.com/sazzad114/KinectEmulator", Code repository for the Kinect emulator.

[8] "https://sites.google.com/site/htibuet/kinect-emulator", The Kinect Emulator Project Page. .