An open-source tool for distributed viewing of kinect data on the web

Bernardo Reis  João Marcelo Teixeira  Judith Kelner
Centro de Informática
Universidade Federal de Pernambuco
{bfrs, jmxnt, jk}@cin.ufpe.br

Abstract—This work presents an open-source tool for viewing the depth map information captured by Microsoft’s Kinect sensor in a distributed way as a 3D point cloud. The goal is achieved by capturing Kinect’s data on the server side and embedding the client application on the web browser, enabling a number of users to access the information at the same time. This work is the first of its kind to perform live manipulation of Kinect’s data on web browsers and as consequence opens a series of future possibilities, from device access sharing to 3D videoconference solutions.

Keywords—Kinect; depth-map; remote visualization

I. INTRODUCTION

Microsoft’s Kinect sensor was originally developed to be an accessory for the Xbox 360 video game console, enabling users to control and interact with the Xbox 360 without touching a game controller, through natural postures/gestures [1].

The device functions as a 3D scanner, capable of capturing a depth map of the scene by using a continuously projected infrared structured light, commonly adopted in image-based 3D reconstruction. Kinect’s popularity is due to the fact that its output can be used in a wide range of applications, from robot navigation support [2] to high quality dense 3D reconstruction [3]. More than the output itself are the daily discovered uses with the device in the PC world.

As any other recent technology, it also has its limitations and drawbacks. Because of the fact that it is still in its “baby steps”, its price is almost as high as the one of the Xbox 360 console. Another problem is that since it is connected to the PC via a USB port, a single PC can access the data that comes from the sensor at once. A possible workaround is to use USB-to-network software, which enables the sharing of a USB device over the network. This solution, unfortunately, does not solve the problem of more than one user accessing the device at the same time. Other than that, using the sensor connected to a PC requires the device drivers to be previously installed before executing any application.

Aiming to overcome all the problems mentioned before, this work introduces the KinectWebViewer, an open-source tool that offers capture and dynamic visualization of live 3D data using the Kinect sensor. Through the use of the tool, the information from a single Kinect is shared between different users, at the same time. The application works in a server multi-client scheme, in which a single PC connects to the Kinect and shares its captured data to all connected clients. This way, the driver requirement is particular to the server computer to which the sensor is attached, in a way that the driver installation is no longer an issue. Due to time limitations, this work will not offer any support to audio transmission. This feature will be pointed as future work.

The remainder of this paper is organized as follows. Section 2 lists some works related to the one proposed, specially focusing on the ones that perform transmission/visualization of Kinect’s captured data. Section 3 details the software libraries used in the development of KinectWebViewer, and which was the contribution of each library to the project. Section 4 describes the proposed architecture, along with implementation details for both client and server sides. Section 5 shows the results obtained with the tool and perform an analysis about some specific experiments made. At last, section 5 gives some final considerations on the work done and points possible future directions.

II. RELATED WORK

The works related to the one proposed can be divided into two different classes: the ones focusing representation and compression for transmission over the network and the ones regarding visualization.

In [12], the authors intend to create a depth map representation using a single 2D image as input, and they claim to achieve good quality even with a low complexity algorithm.

In [13], the authors perform depth map estimation and encode the resulting data using regions of interest and a Jpeg-based algorithm, focusing image based rendering.

In [14], a depth map representation for real time transmission and view-based rendering of 3D scenes is proposed. They adapt a triangular mesh simplification technique for depth map representation and achieve moderate compression by encoding vertices in mesh rendering order.

In [15], in order to achieve temporal compression of the 3D video, the authors take into consideration both depth map motion estimation and compensation.

An interesting commercial 3D point cloud viewer made for web browsers can be found in [16]. The visualization of the data is of very good quality, however it is not in real time (every time the point of view is changed, the scene is rendered again by the Adobe Flash application).

A project that unified Kinect’s data capture, transmission and visualization can be found in [17]. They perform the capture of the scene using 5 different Kinect sensors at the same time, combine the depth maps and show the result in full 3D mode to the user, like in a 3D videoconference application. There is no detail regarding the representation, compression and transmission scheme of data adopted by the authors in their work yet.
In the proposed work, a single Kinect sensor is used. The compression scheme initially adopted is the one provided by the network library used. For the visualization, the point cloud is rendered inside the web browser, as a set of point primitives.

III. USED COMPONENTS

The KinectWebViewer uses a set of software components in order to fulfill all pre-requisites proposed in this work. Each one deals with a different characteristic of the application: accessing Kinect’s captured data, distributing the captured data over the network, visualizing the 3D point cloud information and making the application portable enough that it can run inside a web browser. All four components will be briefly detailed as follows.

A. Kinect SDKs

Since its launch in November 2010 until June 2011, there was no official SDK supporting development on the PC side. Through reverse engineering efforts, the developer community was able to hack and interpret most of the data that was transmitted by the USB, and also to better understand how the device works. Well-known libraries are the CL NUI [4], OpenKinect [5] and OpenNI [6], being the last one the most used. Although OpenNI concentrates the majority of examples and can be used with different operating systems, it does not offer any support for controlling Kinect’s tilt motor.

In spite of the fact that Microsoft has released its official non-commercial SDK [7] only in the middle of 2011, supporting skeleton detection and having the simplest installation process (a single executable file contains drivers, SDK and examples). Its main drawback is that up to this moment it only works with Windows 7.

B. RakNet

The RakNet library [8] was chosen for dealing with the transmission of the captured data. It is a cross platform, open source C++ network engine mainly focused on game development. As happens with games, the requirements of KinectWebViewer are the real time transmission of chunks of data, allowing live user interaction. With few lines of code it is possible to establish a connection between server and clients. It also supports different network protocols with an embedded data compression scheme, which turns to be mandatory when transmitting large amounts of data.

C. OpenGL

The OpenGL [9] is used in order to allow the 3D point cloud visualization on client side, due to the fact that it is a well-established platform for supporting rendering of three dimensional data in real time, with direct support from the GPU.

In the developed prototype, the point cloud is rendered as a set of 3D points with a specific size. A better rendering result can be obtained by constructing a mesh that comprises all points in the cloud and then applying a single texture to it.

D. Browser application support

In order to enable the application to run inside a web browser, two libraries were taken into consideration: Mammoth [10] and OSAKit [11]. The first one functions as a C++ Flash streaming server, making possible the network interface between a C++ server application and an Adobe Flash one. The problem with this approach is that server and client applications should be developed for different platforms, and as consequence the programming effort is higher.

OSAKit emerges as a solution the previous problem, in a way that the same executable that runs in Windows can run inside the web browser (in the same operating system) through an ActiveX plugin. This cases the application development and also allows a better performance on the client side, since the application can be developed/executed directly in C/C++.

IV. THE KINECTWEBVIEWER

The possibilities introduced by the development of an easy-to-use 3D scanner as the Microsoft Kinect sensor are vast, however it still isn't an affordable device. The KinectWebViewer is an open-source solution for sharing the sensor's information among multiple users over the web.

The KinectWebViewer was designed as a very simple “multiple clients with single server” application that enables the broadcast of Kinect live data to each viewer. Its simplicity relies heavily on RakNet's productive library and it was completely coded in C++.

The architecture consists of applications: a server that is directly connected to a Kinect sensor and that captures and broadcasts its data; and clients, that can simultaneously access the data through a web browser and manipulate it as a 3D point cloud. Each side of the application is better explained in the following subsections. An overview of the client/server architecture is shown in Figure 1.

![Figure 1. KinectWebViewer client/server architecture.](image_url)

A. Server application

The server interacts with Kinect sensor through Kinect for Windows SDK from Microsoft, acquiring depth map information and color camera images. Although being capable of providing this information in different resolutions, the Kinect SDK’s currently can only align both images when configured to work with color camera images in 640x480 pixels and depth information in 320x240 pixels. Therefore, these are the only resolutions used on the proposed system. The result of the alignment is a 320x240 pixels color image that matches the depth map.
After specifying the alignment of the color image with the depth map, the data is serialized using RakNet functionalities and then sent to each connected client. There is a concern about the amount of data required to be sent, as it totals 375KB for each capture iteration, so later, compression techniques are discussed in order to soften this issue.

The data is sent using UDP protocol, a simple, not reliable and low overhead way of communicating over IP networks. As it is not reliable, it would not be applicable to this work, since no information can be lost when presenting the point cloud. However, RakNet offers built-in retransmission and reordering of packets, turning UDP into a reliable and faster solution than TCP in networks where there is little loss of packets.

B. Client application

The client application is structured as a common OpenGL application, having its input and output operations handled by this library. While not rendering the data on the screen, the application stays ready for any packet that might be sent by the server. It also uses RakNet for receiving the data from the network.

As soon as an entire frame has been received, which comprises both color and depth information, the application triggers the display function, that correctly presents the information as a 3D point cloud.

The 3D point coordinates are calculated using the formulas shown in Equation 1. This way there is a correspondence between the triples \((x_{img}, y_{img}, z_{3D})\) and \((x_{3D}, y_{3D}, z_{3D})\).

\[
x_{3D} = \frac{x_{img} - \text{HALF}_X_{\text{RES}}}{580} \cdot z_{3D} \\
y_{3D} = \frac{y_{img} - \text{HALF}_Y_{\text{RES}}}{580} \cdot z_{3D}
\]

\((1)\)

\(x_{img} \) and \(y_{img}\) correspond to the x and y coordinates of the plane image, which vary from 0 to 320, for the x axis, and from 0 to 240, for the y one. \(z_{3D}\) is given by Kinect and corresponds to the depth of each pixel. \(\text{HALF}_X_{\text{RES}}\) and \(\text{HALF}_Y_{\text{RES}}\) refer to half of the resolution used, being respectively 160 and 120 (for a 320x240 image). 580 is the focal length of Kinect's infrared camera. \(x_{3D}\) and \(y_{3D}\) are the remaining 3D coordinates that are calculated before drawing the points in 3D space. The correspondence from the 2D depth image to the 3D point cloud is shown in Figure 2.

In order to make it available as a web browser solution, the client has been bundled into an OSAKit pack. It is actually the same executable, only embedded in a web page, so it still has good performance, almost like a native application. Figure 3 shows different views of the client application running inside the web browser of three different machines.

The KinectWebViewer is an interesting application that represents the possibilities brought up by the sharing of Kinect sensor information over the web, but any other application that requires that kind of information being shared with multiple users can easily be coupled to the core of this client. The code is currently available for download at http://sourceforge.net/p/kinectwebviewer.

V. ANALYSIS AND RESULTS

In order to evaluate the performance of the proposed tool, some tests were carried out. Each numerical result
presented in this section is the mean of 300 samples of the experiment. All the evaluation was done over a local network (LAN), first because avoiding internet delays, jitter and losses gives more confident data, but also because this is the main target network of the tool.

The machines employed to run the tests were three and for each test it is described which one was used according to the following list: M1) Intel Core i7 960 @ 3.20GHz with 4GB of RAM and a NVIDIA GeForce GTX 480 with 1.5GB of memory; M2) Intel Core 2 Duo E6600 @ 2.40GHz with 2GB of RAM and a NVIDIA GeForce 8800 GTX with 768MB of memory; and M3) Intel Core 2 Quad Q9400 @ 2.66GHz with 4GB of RAM and a NVIDIA GeForce 9800 GX2 with 512MB of memory. All of them use Windows 7.

Measuring the time required to send a frame over LAN with depth and color information acquired from the Kinect resulted in only 118us, as shown in Table I, what can be considered very short when compared with the other steps of the process. It means that, at least over LAN, transmitting the information is not a problem. However, each packet consists of 375KB of data, so it consumes way too much bandwidth, particularly for Internet. As this is a big concern, some effort was made trying to soften it. RakNet has a compression tool built-in, but, although it can compress the packet in about 19%, it takes a long time to do so. This is also shown in Table I. The time to send the packet was calculated as the round-trip time (RTT), or the time to send such packet and receive an acknowledgement for that.

TABLE I. COMPARISON BETWEEN COMPRESSED AND NON-COMPRESSED SENDING USING MACHINE M1.

<table>
<thead>
<tr>
<th></th>
<th>Data size(KB)</th>
<th>Time to compress (in ms)</th>
<th>Time to send (in s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without compression</td>
<td>375</td>
<td>0</td>
<td>118.34</td>
</tr>
<tr>
<td>With compression</td>
<td>304</td>
<td>6494.80</td>
<td>111.45</td>
</tr>
</tbody>
</table>

RakNet compression technique is a per-string Huffman coding [18], but there are better solutions when trying to compress images, as the popular lossy JPEG compression or the lossless compression technique used in the PNG file format. The color image from the Kinect can be compressed with JPEG with no problem, since the consequent noise is masked by the human eye. The depth data cannot suffer from such artifacts, so lossless compression is required. These and other compression techniques are still to be tested.

The whole process was timed also, in order to identify possible optimization points and find out how it would behave on different machine configurations. The resultant profile is shown in Table II, where each step is numbered according to Figure 1. The data capture (1) was timed for how long it takes to grab an unbuffered frame from Kinect. Since Kinect has a maximum of 30 frames per second (FPS), all the tests are consistent with that information. It is important to mention that buffered frames are very quick to grab, in the order of few tens of microseconds. Server-side data processing (2) relates to the procedure of registering color and depth images, which corresponds to a pixel-wise function call to the Kinect SDK, probably performing computer intensive operations, as its performance depends on the machine configuration.

Sending (3) the frames and receiving (4) them is actually very fast, but there should be some degradation when transmitting via internet. The procedure of calculating the 3D point cloud (5) involves iterating sequentially over all 2D pixels. At last, the point cloud is sent to OpenGL so that it can render it on the screen (6). This step is highly dependent on the graphics performance of the machine, thus the newer and more powerful machine M1 shows the best performance.

TABLE II. PIPELINE STEPS EXECUTION TIME (IN MS) FOR DIFFERENT MACHINE CONFIGURATIONS.

<table>
<thead>
<tr>
<th>Step</th>
<th>Machine configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M1</td>
</tr>
<tr>
<td>(1)</td>
<td>20598.66</td>
</tr>
<tr>
<td>(2)</td>
<td>2598.55</td>
</tr>
<tr>
<td>(3),(4)</td>
<td>73.00</td>
</tr>
<tr>
<td>(5)</td>
<td>3154.12</td>
</tr>
<tr>
<td>(6)</td>
<td>1546.04</td>
</tr>
</tbody>
</table>

In order to measure the scalability of the tool, a test was done registering the time to send a frame to multiple users and receive an acknowledgement from everyone. It resulted in an almost-linear function, as shown in Figure 4. Since it behaves linearly, as the number of clients increases, the time for a full cycle of transmission scales accordingly. That’s why a multicast approach may be necessary to avoid a network bottleneck when dealing with a scenario with a high number of users. Further testing based on this new approach is still being conducted.

VI. CONCLUSION AND FUTURE WORK

This paper introduced an integrated multi-user solution for capturing, sharing and viewing Kinect’s data. It directly tackles the problem of Kinect access, from the perspective of number of devices to simultaneous access and application exposure. The performed tests showed that it is possible for a developer to view and manipulate a Kinect’s data stream that comes from the network without considerable effort.

This work opens a series of possibilities regarding Kinect’s use. A simulator for Microsoft’s Kinect SDK, capable of recording the dump of sensor’s data and after reproducing the same depth map would be very useful for testing. Another possibility is to improve Microsoft’s SDK with the ability to connect to Kinect devices that are
located on the network, in a way that more than one user can access the same device at the same time. The KinectWebViewer can also pose as an initial idea for a more complex telepresence system for 3D videoconference inside web browsers.

As future directions, more elaborated compression schemes can be used to reduce even more the bandwidth necessary for the data (RGB and depth) transmission. Optimizing the amount of data to be transmitted would allow the authors to research Kinect’s audio transmission as well.

In order to reduce the initial loading time for the web browser application that is distributed, it is possible to reduce the dlls’ sizes that are stored inside the OSAKit files. Most of them come from the OpenCV library and just few functions are used by the viewer. At last, the GPU could be used to process in a parallel way the 3D point cloud, enhancing the rendering and increasing application’s FPS.

We also intend to perform tests involving wide area networks, in order to evaluate the delay and its impact on the usability of the system.

REFERENCES