WebGL Based 3D Points
Realtime Visualization from Kinect

Takashi Furuya, Qiong Wang
University of Pennsylvania

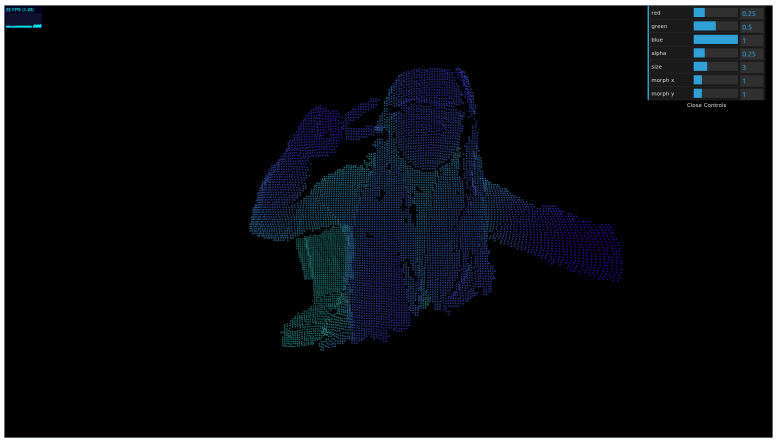

Figure 1. 3D Point Clouds of Depth Value.

Abstract

This paper is the final project report for CIS 565 GPU Programming at University of Pennsylvania. An interactive and realtime 3D point cloud was implemented in the browser. Kinect data was streamed through WebSocket server to the browser and rendered using WebGL.

1. Introduction

First, related previous work will be discussed detailedly. Then our project's objective and challenges would be covered.

1.1. Related Work

There has been multiple attempts at displaying Kinect data using WebGL. OpenDepth [Ecaterina Paun and Piturca] is one of the open source projects whose aim is to bridge the gap between Kinect and the browser; however, their cloud point demo is a

snapshot: real-time data is not available for visualization. In constrast, Mr.doob showcases real-time point cloud animation in his Chrome Experiemnts "Kinect+WebGL" [Mr.doob], where he uses Three.js to initially generate points in JavaScript and streams in WebM depth video for use as texture. These two are then combined to calculate the 3D point location in the vertex shader. However, this method requires preprocessing the depth data into a video format. Microsoft also ships their Kinect SDK with a Web Application and a JavaScript library; however, it is left for the developer to extend it to stream and display 3D point cloud. Another open source project adds streaming and visualization of the video camera and displays real-time point cloud in the browser[Elespuru], and this is accomplished by first downsampling the 3D vertex positions and then applying LZMA compression in the server. Finally, it is de-compressed in the browser (via JavaScript) which is fed into Three.js's particle system. In almost all implementations, a WebSocket server is involved in streaming Kinect data to the browser. We investigate the feasibility of creating a real-time WebGL 3D point cloud visualization from the Kinect by implementing a similar WebSocket system from close to scratch and analyzing its potential bottlenecks and limitations. In addition, we extend Elespuru's project to create visually attractive demonstrations to showcase what is possible with Kinect's real-time point cloud data.

1.2. Objective and Challenges

As of this writing, there are no WebGL demos showing real-time 3D point cloud visualization with appropriate RGB color from the camera. To display true RGB value in the 3D point cloud space, we need to transmit both the depth and the RGB value at the same time. Our objective is to investigate how feasible it is to accomplish this. It would be great to be the first one to visualize realtime true RGB as point cloud based on WebGL. However, it is difficult to synchronize both depth data and RGB data and process all of this from the server to the client.

2. Project Environment and Implementation

We created two projects, one in Linux (Ubuntu 12.04 LTS) and the other in Windows (7 SP1). The Windows project aims to analyze performance while the Linux project extends Elespuru's work to create an interactive cloud point demonstration. Both projects were developed and tested with Google Chrome 31.

In both cases, a program runs in the background on the computer with the browser. This program connects to the Kinect and streams the data to the browser as WebSocket server through localhost, as shown in Figure 2. This data is received in the browser and is rendered as 3D graphics on an HTML canvas element via WebGL.

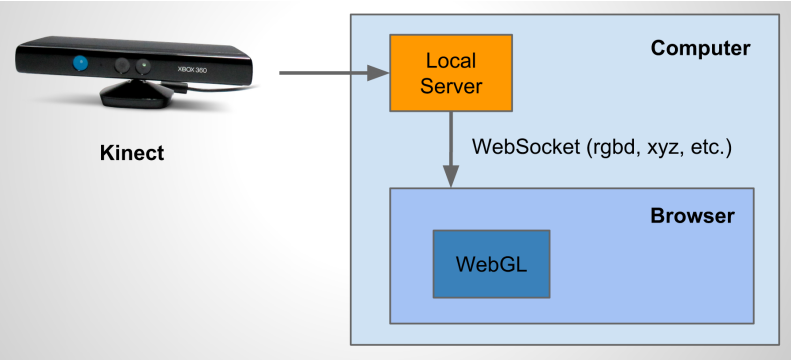

Figure 2. System to stream Kinect data to browser uses a local WebSockets server.

2.1. Windows

The server is implemented in C♯ using Alchemy [Olivine_Labs] WebSockets server library and Windows Kinect SDK. In order to ensure that high performance is achieved, we implemented the client with the following guidelines:

- Redundant WebGL calls should be avoided especially in rendering loops.
- Buffer should be passed to the GPU efficiently (e.g. using TexSubImage2D instead of TexSubImage2D to update texture without memory re-allocation).
- Manual copying/moving of data or looping over large buffers using JavaScript should be avoided by relying on the browser's native data processing/passing methods.

In order to ensure that the above criteria are met and to have greater control over our implementation, we wrote most of the code from scratch without relying on third party 3D libraries such as Three.js.

2.2. Linux

The server part on Linux System is directly folked from Elespuru's project. The main improvement is in the client part to visualize the 3D point clouds. To run the python codes of websocket in Linux, we need to operate the following commands in terminal so as to build the libfreenect library.

```
cd libfreenect/wrapper/python
cmake CMakeLists.txt
make
sudo python setup.py install
```

After installing the python compiler correctly, we also need to download the newest version of easyInstall from the following link
https://pypi.python.org/pypi/setuptools

Extract it in any directory to use the following commands to add the pylzma, autobahn.websocket module.

```
sudo easy_install pylzman
sudo easy_install autobahn
```

After configuration of the environment, we can run the bash script and open the client part webpage to display an interactive 3D point cloud. Here we used a lightweight controller library for JavaScript *dat-gui* [Data_Arts_Team].

In the client part, the following features were accomplished

- 3D point clouds display with pure color with difference of depth;
- 3D point clouds display with control panel to tune the RGBA value;
- 3D point clouds display with deforming along x-axis or y-axis.

In this part, we rely on some existing libraries such as Three.js, stats.js and also detector.js but all the vertex shader and fragment shader are written from scratch.

Additionally, we are still trying to combine the RGB value display with the depth data to visualize the 3D point cloud somehow. After quite a lot trials, we still cannot fix the problem to display the 3D points with both RGB value and depth value. It might well because there is a bandwidth limit for sending and receiving data from a websocket.

3. Performance Evaluation

We conducted four experiments as summarized in Table 1 and compared their performance:

Test No.	**Rendered**	**Content Transferred via WebSocket**	**Transfer Size Per Pixel (Byte)**
1	Depth	Depth	1 (300kB/frame)
2	Color	RGB	3
3	Point Cloud (position calculated on server)	Float XYZ	12 (2.5MB/frame)
4	Point Cloud (position calculated on WebGL)	Depth	2

Table 1. Performance Evaluation

In the first two tests, data obtained from Kinect is directly copied into WebGL texture which is rendered full screen on canvas. In the third test, the 3D position is calculated on the server and passed to the vertex buffer to be rendered directly through a pass-through vertex shader. The last test receives raw depth data (similar to test 1)

Test No.	WebGL Render (FPS)	WebSocket (FPS)	Server (ms)
1	60	30	3
2	60	30	10
3	NA	NA	NA
4	59	30	14

Table 2. Performance Evaluation Results

which is passed to the vertex buffer where its 3D coordinate is calculated in the vertex shader.

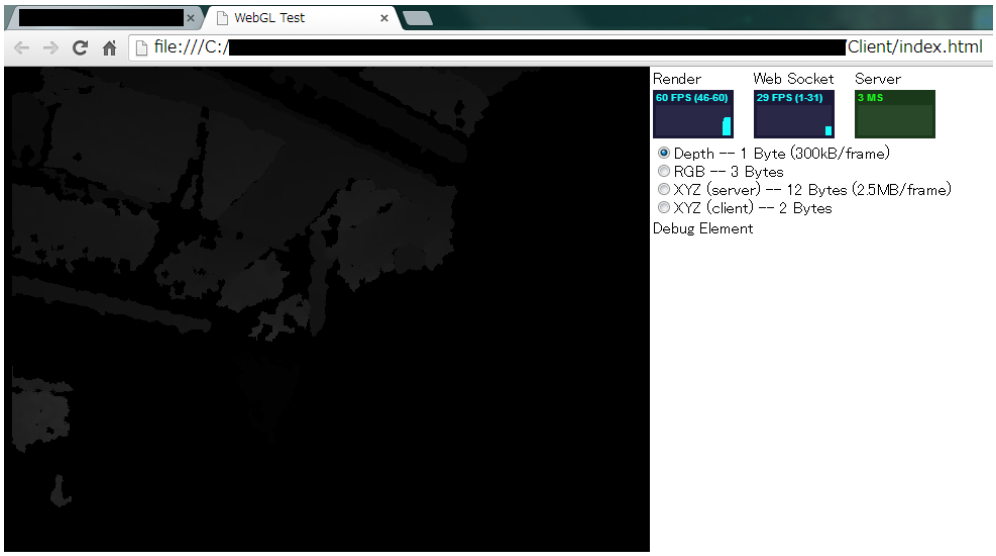

Figure 3. Performance monitor at the top right (Test 1).

In each of these cases, we measured the rendering frame rate, WebSocket refresh rate, and the time it took to process the data on the server (which was not done in parallel) as shown in Figure 3. This last piece of information was stored in the first few bits of the actual buffer data being sent so that the server processing time can be displayed in the browser without affecting the WebSocket bandwidth by introducing extra data transfer overhead.

As shown in 2 all four cases except Test 2 achieved to render at approximately 60 FPS (maximum) and WebSocket of 30 FPS (also maximum since this is the frame rate of data streaming from Kinect). The browser crashed with Test 3. Since the server kept sending packets, this is most likely due to WebSockets bandwidth limit.

4. Conclusion

Rendering Kinect data in the browser with highest resolution at maximum framerate is not as straightforward as passing all of the data through WebSocket and handing it to WebGL. There needs to be some sort of downsampling, compression, or post-processing to be able to render with maximum resolution even if the application is optimized to rely on browser's native data manipulation methods. This explains why there are not demos of real-time 3D cloud point with RGB colors. In our case of rendering 3D points, we determined that calculating the 3D point from the depth on the client WebGL is the way to go to render without losing any detail. We also know that WebSocket becomes the bottleneck of these types of applications.

5. Future Work

Some areas to investigate are effectiveness of JavaScript decompression. Another topic is to use polling or webworker to prevent the browser from crashing due to the server sending too much data over WebSockets.

Acknowledgements

The authors wish to thank Patrick Cozzi for his excelent GPU lectures and valuable WebGL implementation advices especially during the course of this project. We would also like to thank William Boone for helping us closely with every project and making this class go smoothly.

Supplemental Links

Github:
http://bit.ly/1e9obpw
Final Presentation Slides:
http://bit.ly/IT2ZFv
Video Demo:
http://www.youtube.com/watch?v=P4wDHG441ig

References

DATA_ARTS_TEAM. Dat-gui: a lightweight controller library for javascript. 4

ECATERINA PAUN, N. P., AND PITURCA, M. Client server application that brings kinect sdk methods and data to the web. 1

ELESPURU, P. Streaming and visualization of the video camera and real-time point cloud display. 2

MR.DOOB. Experiment visualising kinect data with webgl. 2

OLIVINE_LABS. Alchemy websockets: An extremely efficient c# websocket server for .net projects. 3

Author Contact Information

Takashi Furuya
University of Pennsylvania
3650 Chestnut Street
Philadelphia, PA 19104, U.S.
tfu@seas.upenn.edu

Qiong Wang
University of Pennsylvania
3650 Chestnut Street
Philadelphia, PA 19104, U.S.
qiong@seas.upenn.edu
http://seas.upenn.edu/~qiong