Using Microsoft's Kinect to Generate 3D Object Overlays

Mark Peng
Brendan Tseung

March 19th, 2014
CS 231A

Abstract

Current video call applications often have additional entertainment features, such as visual effects (e.g. hats, facial hair, sparkles) that are overlaid on the video. These visual effects are typically 2D drawings that follow the face as it moves via head tracking. Although the current visual effects are comical and entertaining, not only is the head tracking unreliable but also the 2D drawings are obtrusive and awkward looking. We present a method using the Kinect for overlaying 2D objects and 3D objects that track a user in all 6 degrees of freedom. Experiments have shown that with the open source development environment and API chosen (OpenNI and NiTE) that overlaying objects realistically on a user is indeed possible. However, there's still a need for expanding the API of the open source software in order to have full 3D object overlay functionality on the Kinect.

Introduction

Google Hangouts is a video call application that allows users to overlay 2D objects onto users. These 2D objects can range from hats and glasses to devil horns and halos. The overlays track the user's movement throughout the duration of the video call. Unfortunately these 2D overlays only track the rotation of the user along the camera axis, causing the overlay to be not as realistic as it should be as shown below in **Figure 1**.

Figure 1 Google Hangout Overlays only Rotate about the Camera Axis

Because of this limitation, this project focuses on the generation of 2D and 3D visual effects that appear to be placed in 3D space in the video. These 3D visual effects would blend into the video, and by translating, rotating, and scaling proportionally to objects in the video, they would actually seem to be merged in with real world objects. More realistic and immersive visual effects could lead to higher entertainment value. But more importantly, users might actually be able to convincingly customize their appearance during video calls.

Related Works

Current technologies that are similar to what we are trying to achieve actually superimpose 3D objects onto video. Our system differs in that the 3D visual effect will be calculated, transformed, and rendered in real time. Research has been conducted on 3D reconstruction using a depth camera [1] as well as 2D image overlays on video [2][3]. We hope to leverage research in both areas to construct our system.

Installation of Development Environment (OpenNI + NiTE API)

Since accessing the Kinect's depth data is difficult without having the proper development environment from Windows, the decision to go with open-source software was natural. However, the risk with open-source software is that it can be sometimes be a source of pain to install correctly, so this section will give a brief overview of how our development environment was set up in order to leverage the open source packages for the project.

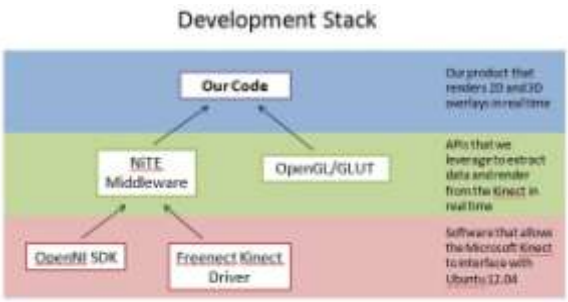
Our project was developed on Ubuntu 12.04 with the following packages installed:

1. GCC 4.x
2. Python 2.6+/3.x
3. LibUSB 1.0.x
4. FreeGLUT3
5. Doxygen
6. GraphViz
7. Freenect

It was then necessary to install the OpenNI and NiTE APIs. We chose to install the newest OpenNI SDK (2.2.0.33 Beta x64)[1] and the newest version of the NiTE Middleware (2.2.0.11)[2]. Given that both of

[1] http://www.openni.org/openni-sdk/
[2] http://www.openni.org/files/nite/

these libraries were originally intended for use with the PrimeSense Carmine depth camera, an additional library, libfreenect[3] (part of the OpenKinect open source project), was leveraged for its Microsoft Kinect driver. To test the installation and to compile the OpenNI and NiTE sample code, it was necessary to modify the provided Makefiles and make sure that GCC was finding all the libraries for linking. In order to run the executables, the driver Freenect (from libfreenect) needs to be copied to the /bin directory of the OpenNI examples.



Once installed, OpenNI and NiTE provide skeleton tracking capabilities, which gives the position and orientation data of various joints in the body as shown in **Figure 2**.

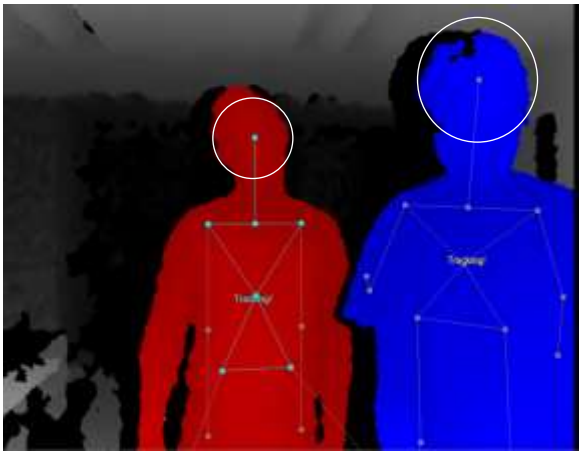


Figure 2 Skeleton Tracking from OpenNI and NiTE API

Because of the massive amounts of data provided by OpenNI and NiTE, it was definitely worth the time and effort into installing these packages.

Rotations of 3D Objects from Quaternion Data

Orientation data provided from the NiTE API was given in terms of rotation quaternions, which are in a more compact form than Rotation Matrices and have the benefit of not having singular values at certain orientations. A quaternion has the following form:

$$\bar{\boldsymbol{q}} = [w \;\; x \;\; y \;\; z]^T$$

Recall the Euler Angle-Axis representation for a rotation, which states that any rotation may be expressed as an angle about a rotation axis. The rotation angle was given by θ and the rotation axis unit vector was given by $\hat{\boldsymbol{e}}$.

The elements of a rotation quaternion can be expressed in terms of the Euler Angle-Axis representation:

$$w = \cos\left(\frac{\theta}{2}\right)$$
$$\hat{q} = [x \;\; y \;\; z]^T = \frac{\hat{\boldsymbol{e}}}{\sin\left(\frac{\theta}{2}\right)}$$

A rotation quaternion given by OpenNI is also a unit quaternion

$$\|\bar{\boldsymbol{q}}\| = 1$$

Because of the fact that we have a unit quaternion, we can directly extract the axis of rotation and the angle of rotation by rearranging the above equations, which yields:

$$\theta = 2\cos^{-1}(w)$$

Now that we know the rotation, we can then solve for the axis of rotation:

[3] https://github.com/OpenKinect/libfreenect

$$\hat{\boldsymbol{e}} = \sin\left(\frac{\theta}{2}\right) \cdot \hat{q}$$

The reason we choose the axis-angle representation over the standard rotation matrices is because of the fact that we're using OpenGL and GLUT (Open**GL** **U**tility **T**oolkit) to render our video stream and overlays. The OpenGL packages use the axis-angle representation in order to rotate 3D objects, and by extracting this data from the Kinect, we can easily rotate overlay objects that we generated by using the OpenGL library package.

Creating 2D Object Overlays

The NiTE API, as discussed in the sections above, automatically provides joint coordinate data in "world" coordinates, which it gets from the Kinect's depth data.

$$X_w = [x_w \quad y_w \quad z_w]$$

The world coordinates then can be converted into pixel coordinates using a 2D projective transformation that comes with the NiTE API. A simple function call allows us to pass in world coordinates and get pixel coordinates. Although we didn't look "under the hood", it is assumed assuming that NiTE utilizes some sort of camera matrix, *M*, that was calibrated (we saw that before tracking objects it takes time to "calibrate").

$$X_{pixel} = K[R \quad T]X_w = MX_w$$

Using the pixel coordinates of the different joints, we're able to display 2D objects in the correct location. Since joint data is updated in real time, we can also have the 2D objects track head movement in real time. This takes care of the translation portion of the image overlay.

Even though we have the projective transformation in 2D, OpenGL actually takes 3D "pixel" coordinates when rendering objects. So, this means that even though overlays may be constructed in 2D, after rotation the objects have a certain depth that needs to be accounted for when rendering in the window. If the depth is not properly accounted for, then OpenGL will cut off parts of the object that go past a depth threshold.

For our 2D overlays, we constructed 2D objects in OpenGL (but still with 3D vertices) and then used the rotation data from the quaternions to rotate our overlay correctly by using OpenGL's rotate function with the angle-axis representation. The results of overlaying 2D images using the above method are given in **Figure 3**.

Figure 3 2D Image Overlays with Scaling (top) and Rotation (bottom)

The results show that the method was successful in overlaying a 2D object that and transforms them correctly through scale, rotation, and translation. From the method presented in this section, we have

successfully replicated Google Hangout's overlay feature, while extending it to include rotation in 2 more axes.

3D Object Overlay Issues

In theory, the above method should also work for 3D objects, since OpenGL renders in 3D "pixel" coordinates, given that you specify the depth data correctly. In order to figure out the issue, we used a 3D cube overlay. **Figure 4** shows how our method displays the 3D cube overlay, which colors each of the faces a unique color.

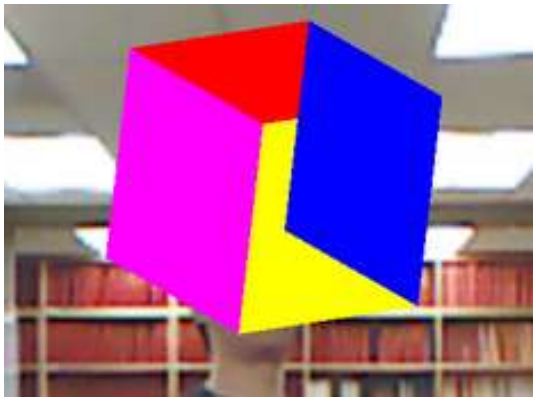


Figure 4 Overlay of a 3D Cube

From the figure above, it is clear that the rendering of the front face is not being performed correctly. In order to troubleshoot the problem further, we display only the front face of the cube as shown in **Figure 5**.

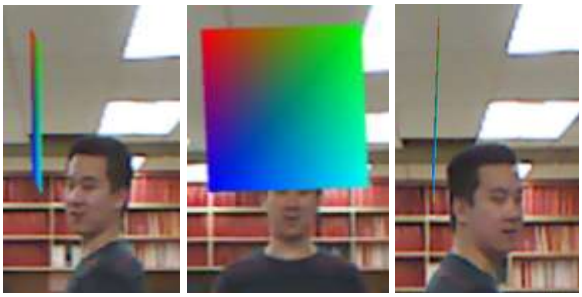


Figure 5 Front Face Cube Overlay while Rotating (left to right)

After rotating 180 degrees, the front face of the cube flips to the back of the head. This means that we're not getting the full 360 degrees of rotation that is actually required for having *true* 3D overlays.

This issue is actually caused by a limitation in the NiTE API and how it provides orientation information. After investigating, it was discovered the head joint orientation was directly correlated to the shoulder joint orientation. If the shoulder joints twisted, then the head joint would receive that orientation regardless of the fact if the head joint actually moved. Furthermore, the left and right shoulder joints are reassigned if the user were to turn around, meaning that we only, in fact, get 180 degrees of rotation information because of the shoulder joint reassignment. This means that we're missing key parts of the orientation data that would allow us to correctly construct 3D overlays.

Conclusions and Future Work

Through our development stack, the Microsoft Kinect was successfully used to track a user in 6 degrees of freedom and to render rotating 2D image overlays in real time. Attempting to extend to rendering 3D image overlays highlighted a core flaw in the NiTE Middleware library; the outputted tracked rotation data from NiTE uses very general heuristics.

Rotation for the head joint is a function of the position and rotation of the two shoulder joints that are tracked. If one of the shoulder joints goes out of scene, e.g. hides behind the other shoulder joint, NiTE recalibrates and reassigns the shoulders. So upon an 180 degree rotation about the y-axis, the user's right and left shoulders would be swapped. This prevents successful rendering of 3D overlays without substantial code on top of the NiTE library.

Future explorations include resolving this weakness in NiTE. This can be

accomplished by reducing the frequency of recalibrations. For example, rather than recalibrating the tracking and swapping the shoulder joints as a result when one shoulder goes out of scene, an function that continuously monitors each shoulder joint can be employed. Thus, through the change in position of the shoulder joints over time, it would be possible to determine which shoulder is still in scene, preventing the incorrect swapping of the shoulder joints.

Further work can also be conducted in improving the fidelity of the overlays, e.g. perfecting the size and positional scaling factors of the overlays and implementing a more user-friendly interface.

References

[1] Izadi, Shahram, David Kim, Otmar Hilliges, and David Molyneaux. "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera." *UIST* (2011).

[2] Uenohara, Michihhiro, and Takeo Kanade. "Vision-Based Object Registration for Real-Time Image Overlay."

[3] Vagvolgyi, Balazs, Li-Ming Su, Russell Taylor, and Gregory D. Hager. "Video to CT Registration for Image Overlay on Solid Organs."