Simulating a 3D Environment on a 2D Display via Face Tracking

NATHAN YOUNG
APRIL 27TH, 2015
EENG 512
Overview

Background on displaying in 3D

Diagram of System

Face and Eye Tracking
  ◦ Haar Classifiers
  ◦ Kalman Filtering

Projecting 3D onto 2D in OpenGL
  ◦ Camera Rotation
  ◦ Camera Positioning
  ◦ FOV Adjustment

Results
Making a 3D display (of sorts)

Current solutions to 3D displays are generally to have two displays, one for each eye.

This can be done by

- Interlacing an image on one display and sending half of it to each eye (as on the 3DS). This does not account for user location and is more of a hologram than VR.
- Using shuttered glasses timed with a high refresh rate monitor to send the correct image to each eye.
- Having two displays, like the Oculus Rift.

In *Mission Impossible: Ghost Protocol*, a different method was used where a camera tracked the location of a guard head and changed the pose of the scene. 

http://youtu.be/ydIPKkjBIMw?t=30s
Oculus Rift’s approach

The Oculus Rift uses onboard accelerometers and screens to immerse the user in a 3D environment. It uses these and its own dimensions to determine what to display. It then adjusts the location of the camera in the world to match the user.

It knows from its dimensions:
- The location of the user’s eyes relative to the screens.
- The height and width of those screens (to calculate aspect ratio)

It will calculate from its sensors and user input:
- The direction the user is looking.
- The location of the player in the world.
My approach

Rather than having a constant distance between the user in the screen, mine will be dynamic.

Instead I will keep the screen in a constant place in the virtual world, similar to a window.

I will use the location of the person relative to the screen to determine what to display on the screen and at what scale and skew to display it at.

I will also use known properties about my computer and camera to determine the location of the person and how to display the image (more on this later).

This means less hardware is needed to track the user, allowing you to trick the user that what they are seeing is real without them having to wear anything.
Diagram of System

Virtual Object

Screen

User
Haar Classifiers\(^1\)

Small identifiers of simple patterns of edges, lines, squares, checkerboards etc. that when 1000’s of them are combined together they form a face.

They must be cascaded in such a way that you start with a simple face and then move to more detailed features.

Does not do well with rotation of any sort.

They must be trained on sets of images with both positives and negatives.

OpenCV has built in Haar Classifiers and trained datasets under its HaarClassifiers class.

---

Haar Classifiers from J Fernandez, I. Wilson [1]
Kalman Filtering

A more robust alternative to Haar Classifiers.

Uses a camera with an IR led in order to light up the eyes and allow for more rigid face dynamics [2]

Starts with a user model for the face and constantly updates it when it changes slightly so that other poses can be found, as long as the user does not quickly move their face.

Can find all three translations and rotations.

Unfortunately I lacked the hardware to attempt this so only Haar classifiers were investigated as they worked well enough.
Detecting a face/eye- Approach

OpenCV has these built-in, trained, and ready to use. (face, eye, left/right eye, etc.)

I used the ones for the face and right and left eyes. I decided to track my dominant eye (right eye) and use it for tracking.

- For long distance 3D simulation, the dominant eye will have a better chance of tricking someone.

As a backup, it would also match the face, and as another backup it would try to match the left eye so it still had something.
Detecting a face - Results

Eyes would get a lot of results (even if you only look for the right one), so I picked the right-most match and called it the right eye.

- I added an option to use left eye dominance in the tracker object.

I also found that the face would not be captured if the head was tilted slightly, making eye capturing more important.
Calculating eye location

I decided to determine the location of my dominant eye (right) relative to the camera.

I determined that the focal length of my camera is 644 pixels, the image captured is 640x480 pixels. I also know my head as captured by the Haar classifier is 7.5”.

From here, I can calculate the z distance of my head to the camera:

\[
\Delta z_{\text{inches}} = \frac{f}{w_{\text{pixels}}} \times w_{\text{inches}}
\]

With that, I can calculate the displacement of my eye relative to the center of the camera.

\[
\Delta x_{\text{inches}} = \frac{\Delta x_{\text{pixels}}}{f} \times \Delta z_{\text{inches}}
\]

These locations could then be used to calculate angles (via atan())
OpenGL and Matrix Transforms

OpenGL is an API for 2D and 3D computer graphics.

- It has many built in functions for building transformation matrices and performing the transforms from sets of vectors. (Rotate, translate, projection, LookAt, etc.)
- It also manages the 3D virtual environment and the locations of all of the objects and their vertices, and does a lot of the math for you.
- Can handle textures, although I have not used them yet.

Using OpenGL, I investigated 3 different ways of simulating a virtual environment.

- Camera Rotation
- Camera Positioning
- FOV Correction
Experiment Parameters

A virtual 4 unit cube centered on the origin of the world

A virtual screen centered 5 units away from the origin along the X of the virtual world (Z axis of real world).

A standard field of view of 45 degrees, unless set otherwise

A standard aspect ratio of 16:9 (the dimensions of my screen) unless set otherwise.

A screen size of 13.5x7.55in

A conversion of 1 unit = 1 inch

Up is located along the Z axis of the screen and the Y axis of the world.
Rotating the Camera

Position the camera at 5 units away from the center of the cube and point it in the same direction the person is looking.

- Direction is found by using the direction the person is looking at.

Good because it keeps a constant distance between the camera and the object and matches the viewing angle. It is also a very simple method.
Diagram of System – Camera Rotation

Virtual Object

User

Screen

θ

45°
Rotating the Camera

Position the camera at 5 units away from the center of the cube and point it in the same direction the person is looking.
  ◦ This is done in OpenGL by using the LookAt function.

Projection is locked at 45 degree FOV and the native screen aspect ratio.
  ◦ This is done in OpenGL by using the perspective function.

Good because it keeps a constant distance between the camera and the object and matches the viewing angle.

```cpp
proj = glm::perspective(
    glm::radians(45.0f),
    float(DISPX / DISPY),
    1.0f,
    300.0f
);

view = glm::lookAt(
    glm::vec3(5.0f, 0.0f, 0.0f),
    glm::vec3(eye.z, eye.x, eye.y),
    glm::vec3(0.0f, 0.0f, 1.0f)
);
```
Results – Can’t see edges of cube, too large
What went wrong?

There is more to simulating a scene than where the camera is pointing

By placing the camera so close to the cube, we made the cube seem much larger than it actually was.

We also could not see as much of the sides of the cube as we would like.
Placing the Camera at the User Location

Position the camera at the calculated locations in inches away from the virtual location of the screen.

Point it at the virtual location of the screen.

This ensures that the camera is pointed in the right direction and the distance the user is away from the screen is accounted for.
Diagram of System – Camera Positioning

Virtual Object

Screen

θ

"45°"

User
Placing the Camera at the User Location

Projection is locked at 45-degree FOV and the native screen aspect ratio.

Position the camera at the calculated locations in inches away from the cube.

Camera is pointed to location of screen this time.

```cpp
proj = glm::perspective(
glm::radians(45.0f),
float(DISPX / DISPY),
1.0f,
300.0f
);

view = glm::lookAt(
 glm::vec3(eye.z + 5.0f, -
 eye.x, -eye.y),
 glm::vec3(5.0f, 0.0f, 0.0f),
 glm::vec3(0.0f, 0.0f, 1.0f)
);
```
Results
Results – Cube shrinks as you move away
Adding FOV Adjustment

The key to making a virtual scene look real is setting the right FOV.

The FOV should roughly match the FOV of the screen relative to the user.

F. Steinke discusses that sometimes you actually have to increase or decrease the FOV depending on what you are displaying. [3]

- Generally objects require a smaller FOV one while environments require larger ones. We are not going to incorporate this.
Diagram of System – FOV Correction

Virtual Object

User

Screen

θ

w

w'
FOV adjustment

Takes place in two parts.

The first part calculates the apparent width and height of the screen based on the angle the user is off of the camera. These are also used to find the apparent aspect ratio of the screen.

\[ w' = w \times \cos(\theta) \; ; \; h' = w \times \cos(\phi) \; ; \; A. R. = \frac{w'}{h'} \]

The second calculates the field of view the apparent width takes up for the user. \( \theta \)

\[ fov = 2 \times \tan^{-1} \left( \frac{0.5 \times w'}{z} \right) \]
Adding FOV Adjustment

Set the field of view to the field of view for the screen

Set the aspect ratio to the apparent aspect ratio that the user sees (prevents warping)

Point the camera 5 units away from the center of the cube.

Place the camera where the user is located relative to the cube.

```cpp
proj = glm::perspective(fov, wp / hp, 1.0f, 300.0f);

view = glm::lookAt(
    glm::vec3(eye.z + 5.0f, -eye.x, -eye.y),
    glm::vec3(5.0f, 0.0f, 0.0f),
    glm::vec3(0.0f, 0.0f, 1.0f));
```
Results
Demonstration of FOV corrected video
How did these three methods compare?

The third method clearly worked the best.
- It was based on the real geometry of the scene and kept into account the FOV of the user as well as their location.

The other two scenes ignored certain parameters of where the user was located, which made them look bad.
Some limitations of the test

The camera was limited to a 53° FOV (26.5° in either direction)

- This FOV is actually a little smaller because it had to detect the face in order to work properly.

Haar Classifiers were not as robust as I’d like and resulted in a lot of skipping, if I were to do this again I would go straight to Kalman filtering.

The model was not very realistic and lacked lighting and texture. And as a result...
Did it look 3D?

No, it really didn’t, the effect was interesting but nowhere near 3D.

◦ The model used was unrealistic and had no texture.
◦ The screen used was matte and not glossy, making it obvious that it was a screen, not a window.
◦ I was displaying the same image to both eyes (not stereoscopic) and my screen was too close for one image to work.
Next steps/Future work

Render more realistic models to determine how real the scene looks.

Test on a larger screen and simulate a larger environment (like a hallway or room).

Use Kalman filtering to improve face tracking and allow for head tilt.
Sources


Code Sources

Libraries used:
- OpenCV
- OpenGL 4.3.0
- SFML 2.2

Code Resources
  https://open.gl/