ECSE 4961 Project 1

Aidan Westphal

October 2023

Purpose

The purpose of this project was to perform full perspective projection on a provided set of coordinates and surface normals. We would read in the data tables, execute the projection, and generate a visualization of the result using light radiometry.

Theory

Cameras produce images by sensing light that has reflected off of an object's surface and through its lens. Generally, there will be an array of photoelectric cells (such as those found in a CCD camera) that recognize this light and convert it to electrical charge, yielding analog images.

We often generalize camera models to the pinhole model, which reduces the camera's aperture (the size of the lens which light can pass through) to a point. This leads to the property that points in the camera frame have a 1 to 1 correspondence to points in the image plane.

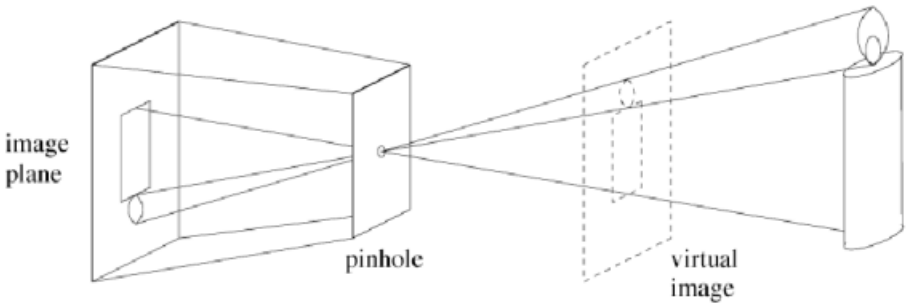


Figure 1: A pinhole camera leads to a 1 to 1 correspondence between the camera frame and image plane.

In the above image, we also see that images are inverted in the image plane. This is known as the real image. As such, we often use the virtual image plane instead. More on this later.

There are several transformations that need to happen to convert arbitrary points into a perspective image:

We first have the object frame, which describes the 3D coordinates of the object. We now recognize that the camera is located somewhere in this object frame and, as such, the coordinates in the object frame need to be described in terms of the camera's positioning. We need to map the origin of the object frame to the position and orientation of the camera's position (where the "pinhole" is). From here, we need to calculate the points in the image frame (recall that pinhole cameras result in a 1 to 1 correspondence between points in the camera frame and image plane). The last step is to account for pixel density and translate points in the image frame to a computer-friendly coordinate system (the origin is a corner and all pixel coordinates are positive).

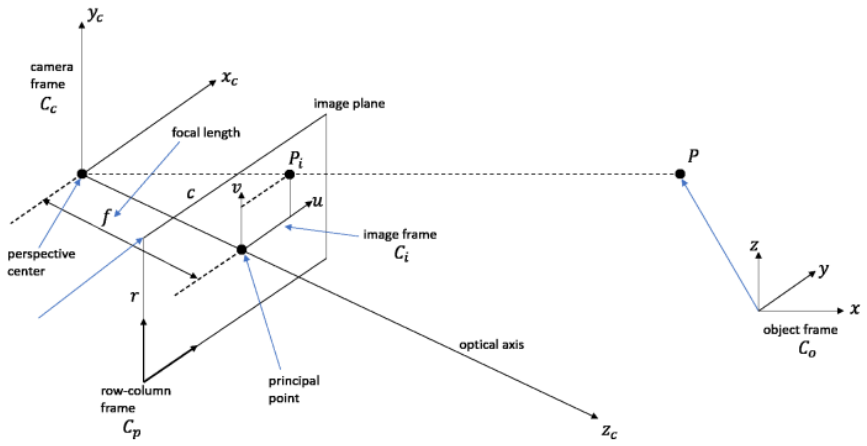


Figure 2: Visualization of the full projective geometry model, including all frames mentioned above.

If the above description sounded vague, it was meant to be. We will now look into the details of each step.

Step 1: Object to Camera Frame We consider a vector $X \in \mathbb{R}^3$ in the object frame and its corresponding vector $X_c \in \mathbb{R}^3$ in the camera frame. Then, the transformation from $X \to X_c$ can be described by a rotation and

translation.

$X_c = RX + T$ for a rotation matrix R and translation vector T.

Step 2: Camera to Image Frame The conversions between the camera and image frame rely on some simple trigonometry which I have chosen to draw out on paper.

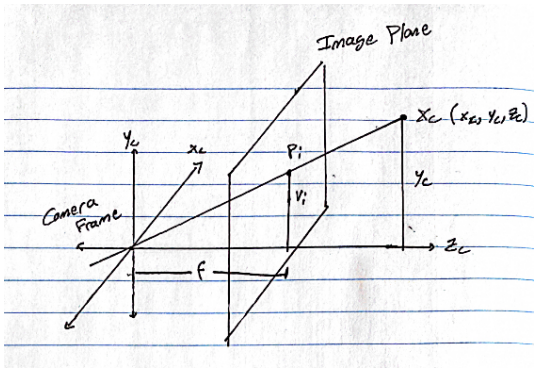


Figure 3: The relation between the camera and image frames is a simple trigonometric relationship.

We note the ratio $\frac{y_c}{z_c} = \frac{v_i}{f} \implies v_i = \frac{fy_c}{z_c}$. We do the same calculation for x_c, yielding the following relationships between the frames:

$$u = \frac{fx_c}{z_c} \; v = \frac{fy_c}{z_c}$$

Step 3: Image to Row-Column Frame We need to translate by the coordinates of the principal point (where the origin gets mapped to in the image frame) and scale by the pixel densities sx and sy. We can represent this step in the same matrix equation format as transfoming the object frame to the camera frame.

$$\begin{bmatrix} c \\ r \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & sy \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} c_0 \\ r_0 \end{bmatrix}$$

Using homogenous coordinates, steps 2 and 3 can be combined by letting $\lambda = z_c$ and using the equation

$$\lambda \begin{bmatrix} c \\ r \\ 1 \end{bmatrix} = \begin{bmatrix} s_x f & 0 & c_0 \\ 0 & s_y f & r_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix}$$

Which leads to the following full perspective projection:

$$\lambda \begin{bmatrix} c \\ r \\ 1 \end{bmatrix} = \begin{bmatrix} s_x f & 0 & c_0 \\ 0 & s_y f & r_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

where [R T] is the Affine Transform with a rotation matrix R and translation vector T. For programming it's worth noting the distinct steps in this calculation as I interchangeably use forms at times.

Radiometric Equation

Once we have the pixel interpretation of our object, we need to decide how each pixel will be colored or, in other words, how light would reflect off this point and into the pinhole camera. This is where the radiometric equation comes into play.

We make the assumption that we have a Lambertian Surface, or in other words, incident light is scattered in all directions upon impact. This behavior aligns best with rough surfaces where light can reflect around on coarse materials. The power of light emitted from the 3D point, the scene radiance R, is found by $R = \rho L \cdot N$ for L being the direction of incident light, N being the surface normal, and ρ being the surface albedo.

From here, we can calculate the power of the light received by the actual camera, the image irradiance, by the following: $E = R\frac{\pi}{4}(\frac{d}{f})^2 \cos^4 \alpha$ where d is the lens diameter, f is the focal length, and α is the angle formed between the point and the camera which can often be assumed to be 0 for small angular aperture or far objects.

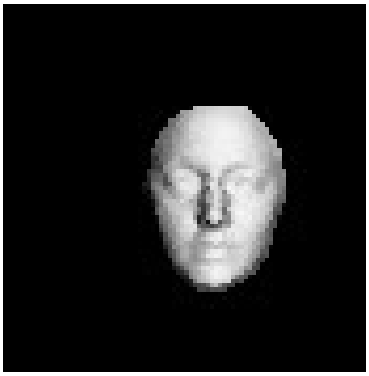
The final step to get image irradiance is to multiply by a coefficient β which depends on camera and frame grabber settings, so combining the above we get the Fundamental Image Radiometric Equation:

$$I = \beta \frac{\pi}{4}(\frac{d}{f})^2 \cos^4 \alpha L \cdot N$$

In the program, we are provided with all of these parameters.

1 Results

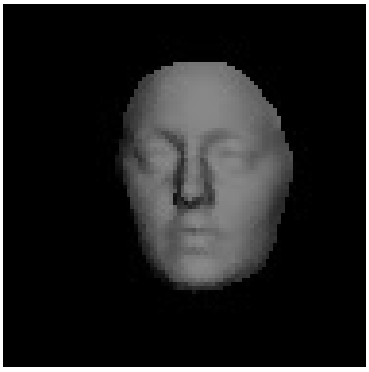
Full Perspective:



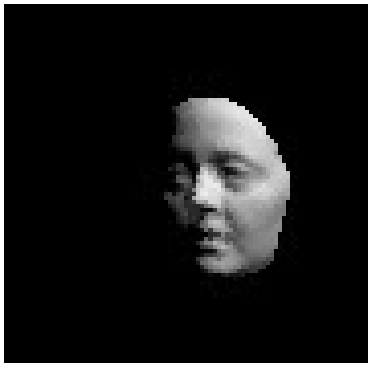
(a) R1, L1, F30 Full Perspective

(b) R2, L2, F40 Full Perspective

Weak Perspective:

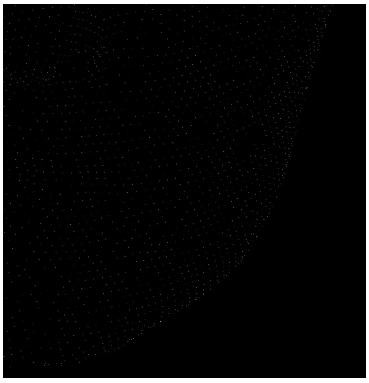


(a) R1, L1, F40 Weak Perspective



(b) R2, L2, F30 Weak Perspective

Orthographic Perspective:

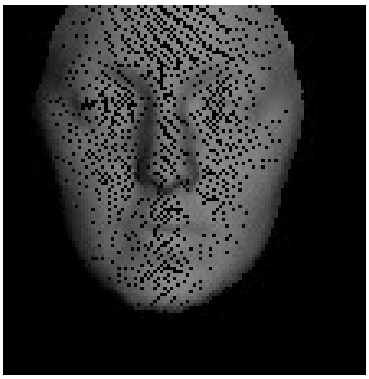


(a) R1, L1, F30 Orthographic Perspective

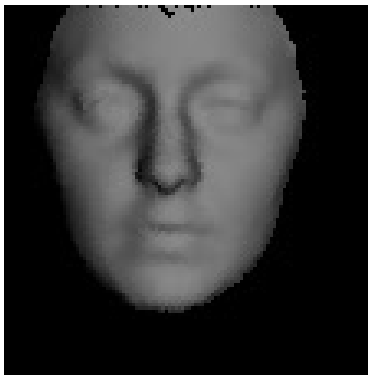
(b) R1, L1, F5 Orthographic Perspective w/ sx, sy = 1

I'd like to note the strange behavior found from the orthographic projection. I belive this is due to the fact that the face is not scaling down with distance from the camera. As such, the pixels that compose the face are sparse. I was able to make said pixels more clear by raising sx and sy to 1 pixel/mm. Note that the images here are very large. We can't even see the pixels in the left image despite them being the same size as all other pixels in this program.

I also implemented a hole-filling algorithm that averages out any holes enclosed by other pixels.



(a) Before Filling



(b) After Filling

Note the existence of some holes on the top of the face. That's because these pixels are attached to the border. The algorithm I implemented uses

adjacency to the border (being "open") as the deciding factor for whether a hole should be filled in or not. If the hole is fully enclosed by nonzero pixels, its interior will be set to the average of the pixel intensity of all bordering pixels.

Summary

In the end, I'm happy with my results. I will say that this took more time than I anticipated, mostly due to me implementing more than I needed to, but it was fun. As mentioned above, the two main issues I have involve my filling algorithm and the orthographic projection, both of which being things I don't think I can fix. For example, I can't do much about the fact that orthographic projection yields massive images. Similarly, I can't do much about my fill algorithm if its logic uses the aforementioned "closed hole" condition. The main solution would be to make a more advanced fill command, but I would prefer not to do this.

The other interesting thing was that my light direction is not exactly what was found in the presentation. I spent quite some time trying to fix this, such as rotating the normal and light vectors, but I quickly realized that this was redundant as the inner product remains constant over the same transformation of those vectors. It makes sense to me that the light direction should be a spatial function rather than a camera function (i.e. the light direction should not be relative to the camera's positioning). It also appeared to me that, in the images from the slideshow, the light still seemed to be linked to the camera rather than space. Shedding some light on this would be greatly appreciated.

Other than error and discussion of my code, I would like to mention some small specifications of how I wrote this. I converted the .mat file to a .csv using MatLab and programmed everything in C++ using a custom Matrix class/data structure. I then reused the write_pgm function from HW1 to write my finished image to a pgm file. To convert the pgm images to jpgs for display in this writeup, I used ffmpeg.