3D Computer Vision

Project #1: Single-View AR Cube

October 22, 2025

Amirkabir University of Technology
Computer Engineering Department

LLM Usage Policy Notice
We have used GPT to help refine this project description! You may use AI tools for assistance as well, but you must be able to explain every line of your code and may be asked to implement parts from scratch in a face-to-face meeting with the TAs.

Objective

Implement a complete single-view geometry pipeline that inserts a perspective-correct 3D cube into one photograph of a planar surface. From a single image and a few manual clicks, you will estimate two orthogonal vanishing points, recover a simple intrinsic calibration (focal length with centered principal point), compute a planar homography, decompose it to obtain the camera pose with respect to the support plane, and finally project and render a cube aligned to the scene.

Data and Tools

Use a single photograph that clearly contains a dominant planar support surface (e.g., tiled floor, desk, wall poster) with long lines. You may capture your own photo or use a public one. Implement in Python with OpenCV/NumPy.

1. **Image Choice and Manual Annotations**
   Choose a suitable image and justify your choice. Annotate four corner points of a planar quadrilateral and at least four line segments in two orthogonal directions. Save all coordinates and show an overlay visualization.

2. **Vanishing Points, Horizon, and Intrinsics**
   Compute the vanishing points for the two in-plane directions by intersecting the corresponding clicked line pairs; report the two vanishing points in homogeneous image coordinates and draw them on the image along with the back-projected lines.

The horizon is the line through these vanishing points; overlay it as well. Assuming zero skew, square pixels, and a principal point at the image center, estimate the focal length f using the orthogonality constraint $v_x^\top \omega v_y = 0$ with $\omega = K^{-\top}K^{-1}$. Form

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

report the numerical value of f, and visualize the principal point and horizon for reference.

Bonus: Automatic Surface Detection
Implement automatic surface detection instead of manual annotation. Use edge detection (Canny) and line detection (Hough transforms) to automatically identify planar surfaces, detect parallel lines, compute vanishing points, and estimate surface boundaries. Discuss the challenges and benefits of this automated approach compared to manual clicking.

3. **Planar Homography and Pose Recovery**
Define the world plane as $(X, Y, 0)$ and map the four clicked image points to a canonical unit square with corners $(0,0), (1,0), (1,1), (0,1)$. Estimate the plane-to-image homography H using normalized DLT; show reprojection of the four corners and report the mean pixel error. Recover the camera pose relative to the plane from $\tilde{H} = K^{-1}H = [\tilde{h}_1\ \tilde{h}_2\ \tilde{h}_3]$ by setting

$$r_1 = \frac{\tilde{h}_1}{\|\tilde{h}_1\|}, \quad r_2 = \frac{\tilde{h}_2}{\|\tilde{h}_2\|}, \quad r_3 = r_1 \times r_2, \quad R = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix}, \quad t = \frac{\tilde{h}_3}{\lambda},\ \lambda = \frac{\|\tilde{h}_1\| + \|\tilde{h}_2\|}{2}.$$

Orthonormalize R (e.g., via SVD) and choose signs to satisfy cheirality (the plane is in front of the camera). Provide a figure that draws the recovered plane axes projected into the image.

4. **Cube Definition, Projection, and Rendering**
Place the base of the cube inside the unit square on the plane; use an edge length s (you may take $s = 1$). Construct the cube so that its base lies within this square and its height extends perpendicularly from the plane. Form the projection matrix $P = K\,[R \mid t]$, project the cube vertices, and draw a clean wireframe overlay. Then render filled faces with simple depth ordering (a painter's algorithm suffices). Submit two images: wireframe-only and shaded faces. If any vertices approach the horizon causing instability, briefly note the observation.

5. **Face Shading Implementation**
Implement proper shading for each face of the cube using Lambertian reflectance model. For each face of the cube, calculate the outward-pointing normal vector using the cross product formula:

$$\vec{n} = (\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)$$

where $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$ are three vertices of the face in counter-clockwise order. Normalize the normal vector:

$$\hat{n} = \frac{\vec{n}}{||\vec{n}||}$$

Apply Lambertian shading to compute the brightness of each face:

$$I = I_0 \times \max(0, \hat{n} \cdot (-\vec{l}))$$

where $\vec{l}$ is the normalized light direction vector, I_0 is the base intensity, and $\cdot$ denotes the dot product. Include ambient lighting to prevent complete darkness:

$$\text{brightness} = \max(0.2, \hat{n} \cdot (-\vec{l}))$$

For each face i with normal $\hat{n}_i$, compute the final color as:

$$\text{color}_i = \text{base\_color}_i \times \text{brightness}_i$$

Include a figure showing the cube with proper face shading. Discuss the choice of light direction and its effect on the visual appearance of different faces. Provide the computed normal vectors and brightness values for all six faces of the cube.

Submission

Submit a concise PDF report (3–5 pages) containing: a description of the method and equations you used, clear figures for each stage (annotations, vanishing points and horizon, rectified plane, wireframe and shaded cube with proper face lighting), and the discussion. Provide runnable code (Python + OpenCV/NumPy) and a script or notebook to reproduce all figures.

A Appendix: Shading Model

A.1 Introduction to 3D Lighting Models

In computer graphics, lighting models simulate how light interacts with surfaces to create realistic visual effects. The choice of lighting model significantly affects the visual quality and computational complexity of rendered scenes. This appendix provides a comprehensive explanation of the Lambertian shading model implemented in our AR cube pipeline.

A.2 Physical Foundation: Lambert's Cosine Law

The Lambertian shading model is based on Lambert's cosine law, discovered by Johann Heinrich Lambert in 1760. This law describes how light is reflected from a perfectly diffuse (matte) surface.

Physical Principle: When light hits a diffuse surface, it scatters equally in all directions. The amount of light reflected depends on the angle between the surface normal and the incident light direction.

Mathematical Expression:

$$I = I_0 \cos(\theta) = I_0(\hat{n} \cdot \hat{l})$$

where:

- I is the reflected light intensity
- I_0 is the incident light intensity
- θ is the angle between surface normal and light direction
- $\hat{n}$ is the unit surface normal vector
- $\hat{l}$ is the unit light direction vector
- $\cdot$ denotes the dot product

A.3 Mathematical Derivation

A.3.1 Step 1: Surface Normal Calculation

For any triangular face defined by three vertices $\vec{v}_0$, $\vec{v}_1$, and $\vec{v}_2$, the surface normal is calculated using the cross product:

$$\vec{n} = (\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)$$

Why Cross Product? The cross product of two vectors gives a vector perpendicular to both. Since $\vec{v}_1 - \vec{v}_0$ and $\vec{v}_2 - \vec{v}_0$ are edges of the triangle, their cross product gives the normal to the triangle's plane.

Normalization: The normal vector must be normalized to unit length:

$$\hat{n} = \frac{\vec{n}}{||\vec{n}||} = \frac{\vec{n}}{\sqrt{n_x^2 + n_y^2 + n_z^2}}$$

A.3.2 Step 2: Light Direction Vector

The light direction vector $\vec{l}$ represents the direction from the surface to the light source. For directional lighting (like sunlight), this vector is constant across the scene.

Normalization: The light direction must also be normalized:

$$\hat{l} = \frac{\vec{l}}{||\vec{l}||}$$

A.3.3 Step 3: Dot Product Calculation

The dot product between the surface normal and the negative light direction gives the cosine of the angle between them:

$$\cos(\theta) = \hat{n} \cdot (-\hat{l})$$

Why Negative Light Direction? We use $-\hat{l}$ because the dot product gives the cosine of the angle between the vectors. For lighting calculations, we want the angle between the surface normal and the direction *from* the surface *to* the light source.

A.3.4 Step 4: Intensity Calculation

The final intensity is calculated as:

$$I = I_0 \times \max(0, \hat{n} \cdot (-\hat{l}))$$

The $\max(0, \cdot)$ function ensures that surfaces facing away from the light source (negative dot product) receive zero illumination.

A.4 Painter's Algorithm for Depth Ordering

The Painter's Algorithm ensures correct rendering order by sorting faces from back to front, similar to how a painter layers paint. The key insight is that faces closer to the camera should be rendered after (on top of) faces farther away.

Logic: For each face, we calculate a depth value based on its position relative to the camera. Faces are then sorted by this depth value and rendered in order, ensuring that background faces don't incorrectly appear in front of foreground faces.