

GRIFFITH UNIVERSITY

3006ICT | ROBOTICS AND COMPUTER VISION

PRACTICE BOOK

Chapter 2

Camera Geometry

From 3D Points to Image Pixels and Back

Dr Lei Wang

Griffith University

Student Practice Edition

How to Use This Chapter

This chapter is a practice and reference companion for Workshop 2. It follows the workshop sequence while adding explanations, annotated code, worked numerical examples, visual checks, experiments, self-check questions, and reusable engineering patterns.

Important course-use note

This practice chapter is intended for use within the course. Keep your own code, explanations, experiments, and screenshots in your working folder. Follow the course rules for sharing workshop and assessment materials.

The most important habit in this chapter is:

Think before trusting the output

For every stage ask: What goes in? What comes out? What assumptions are being made? How can I test it? What would a wrong result mean for a robot?

The supplied Workshop 2 builds a camera-geometry pipeline from a 3D camera-frame point to an image pixel, visibility checking, visualisation, and back projection when depth is known.

Chapter Learning Outcomes

- Interpret camera, image, and robot coordinate conventions used in the workshop.
- Explain the pinhole camera projection in plain language and mathematically.
- Construct and interpret a camera intrinsic matrix K .
- Project a 3D camera-frame point into pixel coordinates.
- Distinguish 'in front of the camera' from 'inside the image'.
- Write a reusable projection function rather than repeating equations.
- Visualise projected points and use images as a debugging tool.
- Explain why a pixel alone does not identify a unique 3D point.
- Back-project a pixel when its depth is known.
- Validate a geometric algorithm using forward/backward round-trip consistency.
- Use AI to accelerate implementation while retaining responsibility for correctness.

Workshop Map: What You Are Building

The official workshop asks you to implement Parts 2-5 and submit the corresponding code and results.

Stage	Input	Output	Engineering question
1. Conventions	Coordinate definitions	Consistent frame interpretation	Which direction is positive?
2. Projection	3D point + K	Pixel (u,v)	Does the result make physical sense?
3. Visibility	Pixel + image size + Z	VISIBLE / OUTSIDE / BEHIND	Is the point actually usable?
4. Visualisation	Valid pixels	Image with points	Can I see whether the geometry is right?
5. Back projection	Pixel + depth + K	3D camera point	Can I recover a point consistently?
6. Validation	Original pixel + reprojected pixel	Pass/fail	Did the round trip preserve the measurement?

One subtle point

The workshop title refers to robot coordinates, but the supplied Parts 2-5 primarily establish camera/image geometry. A full camera-to-robot transformation would require an additional extrinsic calibration or a known rigid transform. Do not confuse a camera-frame point with a robot-frame point.

Coordinate Conventions

The workshop uses a camera frame where X_c points right, Y_c points down, and Z_c points forward. Image coordinates u and v also increase rightward and downward, with the origin at the top-left. The robot frame is defined separately: X_r forward, Y_r left, and Z_r upward.



Coordinate-frame warning

A large fraction of robotics bugs are not syntax errors; they are convention errors. Before using a point, always ask: Which frame is this point expressed in?

Quick Coordinate Check

Q1. If X_c increases while Z_c stays fixed, does u move left or right?

Answer: Right, because $u = f_x X_c / Z_c + c_x$ and f_x is positive.

Q2. If Y_c increases while Z_c stays fixed, does v move up or down?

Answer: Down in the workshop's image convention, because v increases with Y_c .

Q3. What is the principal point?

Answer: The pixel (c_x, c_y) , the image location corresponding to $X_c = 0$ and $Y_c = 0$ in the ideal model.

Part 2 - Project One 3D Point

The supplied workshop uses a 640 x 480 image and the intrinsic matrix with $f_x = f_y = 600$ pixels, $c_x = 320$ pixels, and $c_y = 240$ pixels.

```
import numpy as np

width, height = 640, 480

K = np.array([
    [600.0, 0.0, 320.0],
    [ 0.0, 600.0, 240.0],
    [ 0.0, 0.0, 1.0]
])

X, Y, Z = 0.5, 0.2, 2.0

u = K[0, 0] * X / Z + K[0, 2]
v = K[1, 1] * Y / Z + K[1, 2]

print("Pixel:", (u, v))
print("Inside image:", 0 <= u < width and 0 <= v < height)
```

Read these lines carefully

`K[0,0]` is f_x , `K[1,1]` is f_y , `K[0,2]` is c_x , and `K[1,2]` is c_y . The division by Z is the key perspective operation. It is what makes distant points appear closer to the image centre.

Worked Numerical Example

For $P = (0.5, 0.2, 2.0)$:

$$u = 600(0.5)/2.0 + 320 = 470$$

$$v = 600(0.2)/2.0 + 240 = 300$$

Therefore: $(u, v) = (470, 300)$

Physical intuition

The point has positive X and positive Y , so it appears to the right and below the principal point. Its depth is 2 m, so it is in front of the camera.

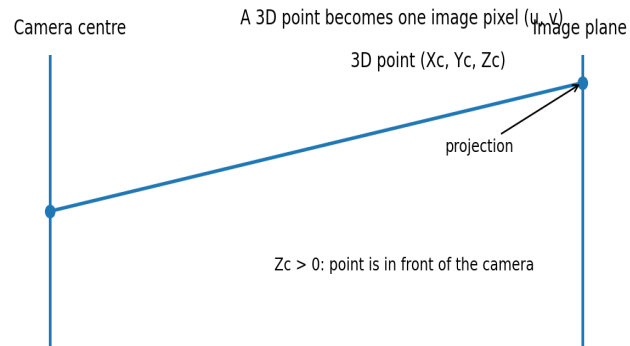


Figure 2. Conceptual pinhole projection. A 3D point is mapped to an image location along a viewing ray.

Predict Before Running

Q1. If Z becomes larger while X and Y stay fixed, what happens?

Answer: The projected point moves toward the principal point (c_x, c_y) . Its displacement from the image centre becomes smaller.

Q2. If $X = 0$ and $Y = 0$, where does the point appear?

Answer: $(c_x, c_y) = (320, 240)$, the principal point.

Q3. If X becomes negative, should the point move left or right?

Answer: Left, because the term $f_x X/Z$ becomes negative.

Q4. What happens if $Z \leq 0$?

Answer: The point is on or behind the camera plane, so the supplied implementation rejects it rather than treating it as a normal visible point.

Part 3 - Build a Reusable Projection Function

The key software-engineering improvement is modularity. Instead of copying the projection equations into every experiment, place them in one reusable function. The supplied solution uses the same function for later stages.

```
def project_point(point_camera, K):
    # Accept a list, tuple, or NumPy array and make it numeric.
    point_camera = np.asarray(point_camera, dtype=float)

    # A point must be in front of the camera.
    if point_camera[2] <= 0:
        return None

    # Homogeneous camera projection:
    projected = K @ point_camera

    # Convert homogeneous coordinates to pixel coordinates.
    return projected[:2] / projected[2]
```

Key Code: What Each Stage Means

Code	What it does	Why it matters
<code>np.asarray(..., dtype=float)</code>	Normalises the input representation.	Keeps calculations predictable and preserves floating-point geometry.
<code>if Z <= 0</code>	Rejects invalid camera-frame points.	Prevents meaningless/unsafe projection.
<code>K @ point_camera</code>	Applies the intrinsic camera matrix.	Compactly represents the camera model.
<code>projected[:2] / projected[2]</code>	Performs perspective division.	Converts homogeneous coordinates to pixel coordinates.
<code>return ...</code>	Returns the geometric result.	Other functions can reuse it without knowing its internals.

Visibility Is a Separate Question

A point can be geometrically projectable but still lie outside the image. The supplied workshop defines visibility using $Z > 0$, $0 \leq u < 640$, and $0 \leq v < 480$.

```
def classify_point(pixel, width, height):
    if pixel is None:
        return "BEHIND CAMERA"

    u, v = pixel

    if 0 <= u < width and 0 <= v < height:
        return "VISIBLE"

    return "OUTSIDE IMAGE"
```

Important distinction

Projection answers: 'Where would this 3D point map in the ideal image?' Visibility answers: 'Does that location lie inside the actual image bounds?'

Test Set

```
points_camera = np.array([
    [ 0.0,  0.0,  2.0],
    [ 0.5,  0.2,  2.0],
    [-0.4,  0.1,  1.5],
    [ 0.3, -0.2,  3.0],
    [ 0.0,  0.0, -1.0]
])
```

Point	u	v	Status	Reason
P0	320	240	VISIBLE	Principal point
P1	470	300	VISIBLE	Positive X,Y; inside image
P2	160	280	VISIBLE	Negative X shifts left
P3	380	200	VISIBLE	Negative Y shifts upward
P4	-	-	BEHIND CAMERA	$Z < 0$

Practice Experiments

- Change P1 from $Z = 2.0$ to $Z = 4.0$. Predict the new pixel before running.
- Change X from 0.5 to -0.5. What should happen to u?
- Change Y from 0.2 to -0.2. What should happen to v?
- Create a point with $Z > 0$ but a very large X, such as (3.0, 0.0, 1.0). Is it in front of the camera? Is it inside the image?
- Create a point with $Z = 0.0$. Does your function reject it safely?

Part 4 - Visualise the Geometry

Visualisation is not decoration. It is a debugging instrument. The workshop asks you to create a blank image, draw visible projected points, label them, display the result, and save the image.

```
def draw_projected_points(points_camera, K, width, height):
    # White RGB/BGR image with shape (height, width, 3).
    image = np.full((height, width, 3), 255, dtype=np.uint8)

    for i, point in enumerate(points_camera):
        pixel = project_point(point, K)

        # Skip points that cannot be used.
        if not is_visible(pixel, width, height):
            continue

        # Geometry stays floating-point until the drawing API needs integers.
        u, v = np.round(pixel).astype(int)

        cv2.circle(
            image,
            (u, v),
            6,
            (0, 0, 255),
            -1
        )

        cv2.putText(
            image,
            f"P{i}",
            (u + 8, v - 8),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.5,
            (0, 0, 0),
            1
        )

    return image
```

Why convert to integers only here?

The projection equations describe continuous geometry. Rounding earlier throws away precision unnecessarily. Only the drawing operation needs a discrete pixel index.

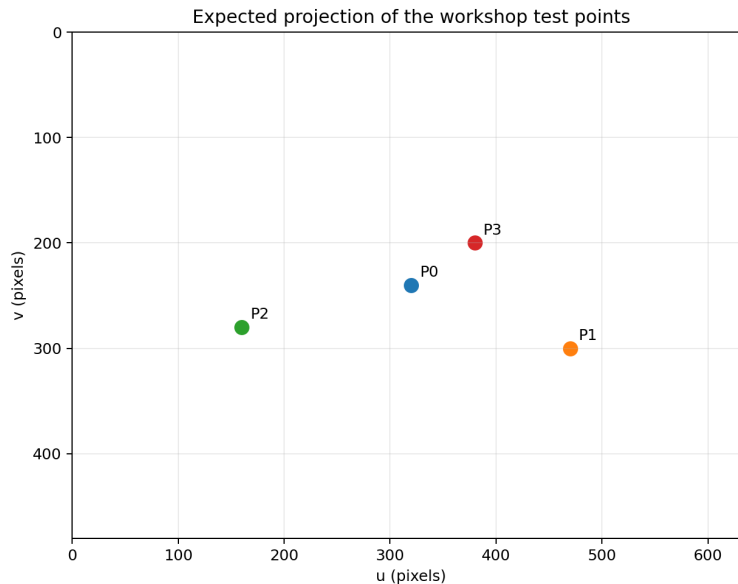


Figure 3. Expected positions of the valid workshop test points.

Nearby vs Distant Points

The supplied solution uses a depth-dependent circle radius as a visual aid. A nearby point can be drawn larger and a distant point smaller. This is useful for intuition, but it is not a physically complete model of projected object size.

```
radius = int(np.clip(14 / point[2], 3, 12))
```

Do not over-interpret the radius

The radius is an artificial visual cue. Real projected object size depends on object size, pose, distance, camera model, and perspective-not simply on the arbitrary drawing radius.

Visual Debugging Checklist

- Is the image size really 640 x 480?
- Does P0 appear at the centre?
- Does a positive X point appear to the right?
- Does a negative X point appear to the left?
- Does a positive Y point appear lower?
- Is the $Z < 0$ point absent?
- If the image looks wrong, did you mix BGR/RGB conventions?
- If a point is unexpectedly outside the image, inspect the numerical (u,v) value before changing the drawing code.

Discussion Questions - Strong Answers

Why must u and v be converted to integers only when drawing?

Answer: OpenCV drawing functions use discrete pixel coordinates, but the underlying geometric calculation is continuous. Keeping floating-point values preserves accuracy until the final rasterisation step.

How could an incorrect focal length change the displayed locations?

Answer: A larger focal length magnifies the displacement from the principal point; a smaller focal length compresses it. Therefore the same 3D point would appear at a different image location.

Part 5 - Back Projection with Known Depth

Forward projection maps a 3D point to a pixel. Back projection reverses the operation only when additional depth information is available. The workshop explicitly emphasises that a pixel alone identifies a viewing ray, not a unique 3D point.

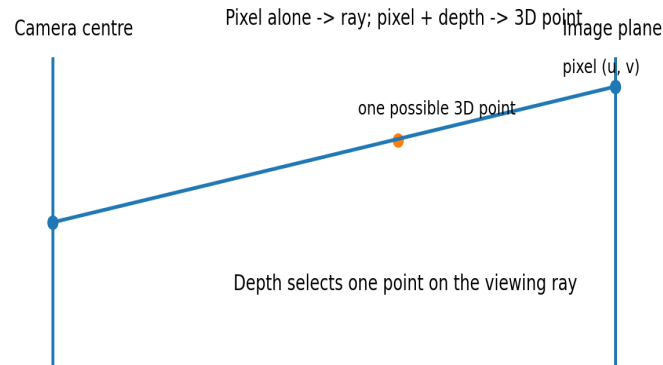


Figure 4. A pixel defines a viewing ray; depth selects one point along that ray.

Inverse Equations

Starting from $u = f_x X/Z + c_x$ and $v = f_y Y/Z + c_y$, rearrange to obtain:

$$\begin{aligned} X &= (u - c_x) * Z / f_x \\ Y &= (v - c_y) * Z / f_y \\ Z &= \text{depth} \end{aligned}$$

```
def back_project_pixel(pixel, depth, K):
    u, v = np.asarray(pixel, dtype=float)

    X = (u - K[0, 2]) * depth / K[0, 0]
    Y = (v - K[1, 2]) * depth / K[1, 1]

    return np.array([X, Y, depth])
```

Worked Example

```
pixel = (470, 300)
depth = 2.0
```

$$\begin{aligned} X &= (470 - 320) * 2 / 600 = 0.5 \\ Y &= (300 - 240) * 2 / 600 = 0.2 \\ Z &= 2.0 \end{aligned}$$

```
Recovered point = (0.5, 0.2, 2.0)
```

Key insight

The recovered point is exactly the point used in the forward-projection example. This is not an accident: the two

operations are designed to be inverse-consistent when the same camera model and correct depth are used.

Round-Trip Validation

Do not stop after obtaining the 3D point. Immediately project it again and compare the result with the original pixel. The supplied solution uses `np.allclose()` for this check.

```
point_camera = back_project_pixel(pixel, depth, K)

reprojected_pixel = project_point(
    point_camera, K
)

print("Recovered 3D point:", point_camera)
print("Reprojected pixel:", reprojected_pixel)
print(
    "Round-trip correct:",
    np.allclose(pixel, reprojected_pixel)
)
```

Engineering habit: validate, don't just calculate

Round-trip validation checks the internal consistency of the forward model, inverse model, and camera parameters. It is especially useful when you do not have external ground-truth 3D measurements.

What If Depth Is Wrong?

- The back-projected X and Y coordinates scale with the assumed depth.
- The reprojected pixel may still match the original pixel because many 3D points along the same viewing ray project to the same pixel.
- Therefore, round-trip pixel agreement validates geometric consistency, but it does not prove that the depth estimate is physically correct.
- This is a crucial robotics insight: validation against one measurement does not necessarily validate every hidden quantity.

Very important distinction

A correct reprojection does not prove correct depth. It proves that the recovered 3D point lies on the viewing ray associated with the pixel under the chosen camera model.

Complete Copy-and-Run Practice Program

The following program combines the core ideas into one small, modular experiment. It is intentionally compact so you can copy it, run it, and then modify one part at a time.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def project_point(point_camera, K):
    # Convert the input to a floating-point NumPy array.
    point_camera = np.asarray(point_camera, dtype=float)

    # Z must be positive: the point must be in front of the camera.
    if point_camera[2] <= 0:
        return None

    # Apply the camera intrinsic matrix.
    projected = K @ point_camera

    # Perspective division -> pixel coordinates (u, v).
    return projected[:2] / projected[2]

def classify_point(pixel, width, height):
    # No pixel means the point was behind/on the camera plane.
    if pixel is None:
        return "BEHIND CAMERA"

    u, v = pixel

    # A point can be in front of the camera but outside the image.
    if 0 <= u < width and 0 <= v < height:
        return "VISIBLE"

    return "OUTSIDE IMAGE"

def draw_projected_points(points_camera, K, width, height):
    # Create a white image.
    image = np.full(
        (height, width, 3),
        255,
        dtype=np.uint8
    )

    for i, point in enumerate(points_camera):
        pixel = project_point(point, K)

        if classify_point(pixel, width, height) != "VISIBLE":
            continue

        # Keep floating-point geometry until drawing.
        u, v = np.round(pixel).astype(int)

        # Optional visual cue: nearer points are drawn larger.
        radius = int(np.clip(14 / point[2], 3, 12))

        cv2.circle(
            image,
            (u, v),
            radius,
            (0, 0, 255),
            -1
        )
```

```

        cv2.putText(
            image,
            f"P{i}",
            (u + 8, v - 8),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.5,
            (0, 0, 0),
            1
        )

    return image

def back_project_pixel(pixel, depth, K):
    u, v = np.asarray(pixel, dtype=float)

    X = (u - K[0, 2]) * depth / K[0, 0]
    Y = (v - K[1, 2]) * depth / K[1, 1]

    return np.array([X, Y, depth])

# -----
# Camera setup
# -----

width, height = 640, 480

K = np.array([
    [600.0, 0.0, 320.0],
    [0.0, 600.0, 240.0],
    [0.0, 0.0, 1.0]
])

# -----
# Test points
# -----

points_camera = np.array([
    [0.0, 0.0, 2.0],
    [0.5, 0.2, 2.0],
    [-0.4, 0.1, 1.5],
    [0.3, -0.2, 3.0],
    [0.0, 0.0, -1.0]
])

# -----
# Projection + visibility
# -----

for i, point in enumerate(points_camera):
    pixel = project_point(point, K)
    status = classify_point(pixel, width, height)

    if pixel is None:
        print(f"P{i}: {status}")
    else:
        print(
            f"P{i}: "
            f"({pixel[0]:.1f}, {pixel[1]:.1f}) - "
            f"{status}"
        )

# -----
# Visualisation

```

```
# -----

image = draw_projected_points(
    points_camera,
    K,
    width,
    height
)

cv2.imwrite(
    "week2_projection.png",
    image
)

plt.imshow(
    cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
)
plt.axis("off")
plt.show()

# -----
# Back projection + round-trip validation
# -----

pixel = np.array([470.0, 300.0])
depth = 2.0

point_camera = back_project_pixel(
    pixel,
    depth,
    K
)

reprojected_pixel = project_point(
    point_camera,
    K
)

print("\nRecovered 3D point:", point_camera)
print("Reprojected pixel:", reprojected_pixel)
print(
    "Round-trip correct:",
    np.allclose(pixel, reprojected_pixel)
)
```

Expected Output

```
P0: (320.0, 240.0) - VISIBLE
P1: (470.0, 300.0) - VISIBLE
P2: (160.0, 280.0) - VISIBLE
P3: (380.0, 200.0) - VISIBLE
P4: BEHIND CAMERA
```

```
Recovered 3D point: [0.5 0.2 2. ]
Reprojected pixel: [470. 300.]
Round-trip correct: True
```

If your output differs

Do not immediately change the equations. First check the input point, K, coordinate convention, image dimensions, and

whether you accidentally converted values to integers too early.

Engineering Practice: Change One Thing at a Time

A useful practice method is to change one parameter, predict the effect, run the program, and explain the result. This turns a code example into an experiment.

Experiment	Change	Predict	What it teaches
E1	Z: 2 -> 4	Point moves toward (320,240).	Perspective scaling.
E2	X: +0.5 -> -0.5	u moves from 470 to 170.	Sign/convention.
E3	Y: +0.2 -> -0.2	v moves from 300 to 180.	Image vertical direction.
E4	fx: 600 -> 300	Horizontal displacement from cx is halved.	Focal length controls magnification.
E5	fy: 600 -> 300	Vertical displacement from cy is halved.	Anisotropic scaling.
E6	depth: 2 -> 4 in back projection	X and Y double for the same pixel.	Pixel alone does not determine depth.
E7	depth: 2 -> 1 in back projection	X and Y halve.	3D position depends on depth.

Mini Challenge 1 - Find an Outside-Image Point

Find a point with $Z > 0$ that projects outside the 640 x 480 image. Record the point and projected pixel.

Your point and prediction:

Your result and explanation:

Mini Challenge 2 - Build a Depth Experiment

Keep pixel = (470,300) fixed and test depths 0.5, 1.0, 2.0 and 4.0 m. Record the recovered 3D points. What stays constant and what changes?

Depth	X	Y	Z
0.5	0.125	0.050	0.5
1.0	0.250	0.100	1.0
2.0	0.500	0.200	2.0
4.0	1.000	0.400	4.0

Observe the pattern

For a fixed pixel, X/Z and Y/Z remain fixed. This is why changing depth moves the reconstructed point along the same viewing ray.

Common Errors and How to Diagnose Them

Symptom	Likely cause	First check	Do not do this
All points shifted together	Wrong cx/cy	Inspect K and principal point	Randomly change X/Y
Horizontal spread too large/small	Wrong fx	Compare u-cx with expected value	Change the 3D point to hide the error
Vertical spread too large/small	Wrong fy	Compare v-cy with expected value	Change image dimensions first
Point moves opposite direction	Sign/convention error	Check X/Y sign and frame convention	Flip signs without understanding why
Behind-camera point appears	Missing Z check	Inspect `if Z <= 0`	Allow division by negative Z
Visible point is not drawn	Visibility or rounding issue	Print pixel and status	Change drawing radius
Image colours look strange	BGR/RGB display mismatch	Check `cv2.cvtColor` before Matplotlib	Change the actual image data
Back projection gives wrong 3D point	Wrong K/depth/equation	Re-derive X and Y from projection	Copy a new formula from memory
Round trip fails	Forward/inverse mismatch	Compare both functions line by line	Ignore the validation result

Five Questions for Any Geometry Function

1. What coordinate frame is the input expressed in?
2. What coordinate frame is the output expressed in?
3. What assumptions must be true for the function to be valid?
4. What are the edge cases: zero depth, negative depth, image boundaries, invalid input?
5. How can I test the function using a known or round-trip result?

Engineering insight

The best test is often one whose expected behaviour you can predict before running. If the output surprises you, investigate the model and assumptions-not just the Python syntax.

Using AI as an Engineering Assistant

This workshop explicitly uses AI as an assistant. The instructor plan emphasises asking AI for focused implementations, checking assumptions, testing edge cases, and improving comments rather than blindly accepting generated code.

Useful Prompts

Purpose	Prompt
Understand the concept	Explain the pinhole camera model in plain English using one numerical example. Define $X, Y, Z, u, v, f_x, f_y, c_x$ and c_y .
Generate focused code	Implement <code>project_point(point_camera, K)</code> using NumPy only. Reject $Z \leq 0$. Return floating-point (u,v) . Explain each line.
Review my code	Review my projection function. Do not rewrite it. Identify logical errors, edge cases, and one test for each issue.
Test the geometry	Suggest five test cases for a pinhole projection function. For each, predict the expected qualitative behaviour before giving numerical values.
Explain a failure	My projected point moved in the opposite direction from what I expected. Give the most likely coordinate-convention causes and show how to test each one.
Validate back projection	Explain how to validate a back-projection function using forward projection and <code>np.allclose()</code> . Also explain what this test cannot prove.

AI rule for this course

If AI generates the code, you should still be able to explain the inputs, outputs, assumptions, important lines, failure cases, and tests. AI output is a proposal-not evidence that the algorithm is correct.

Self-Check

1. What do f_x and f_y represent?

2. What do c_x and c_y represent?

3. Why does the projection divide by Z ?

4. Why does increasing Z move a point toward the principal point?

5. Why can $Z > 0$ still produce an OUTSIDE IMAGE result?

6. Why should projection return floating-point coordinates?

7. Why is $Z \leq 0$ rejected?

8. What information does K contain?

9. Why does a pixel correspond to a viewing ray rather than a unique 3D point?

10. What extra information is required for the simple back-projection function?

11. Why does the back-projection formula multiply by depth?

12. What does `np.allclose()` test in the round-trip experiment?

13. Why does round-trip agreement not prove that the depth is correct?

14. What is the difference between camera-frame and robot-frame coordinates?

15. Why is visualisation useful for debugging geometry?

One-Page Reference Card

Camera model

$$u = f_x * X / Z + c_x$$

$$v = f_y * Y / Z + c_y$$

Back projection (known depth)

$$X = (u - c_x) * Z / f_x$$

$$Y = (v - c_y) * Z / f_y$$

$$Z = \text{depth}$$

Visibility

$$Z > 0$$

$$0 \leq u < \text{width}$$

$$0 \leq v < \text{height}$$

Core reusable functions

- ``project_point(point_camera, K)`` -> 3D camera point to pixel
- ``classify_point(pixel, width, height)`` -> visibility status
- ``draw_projected_points(...)`` -> visual debugging
- ``back_project_pixel(pixel, depth, K)`` -> pixel + depth to 3D camera point

Mental model

Remember

3D point -> projection -> pixel

pixel + depth -> back projection -> 3D point

A pixel gives a direction. Depth gives distance along that direction.

A good engineer does not stop at a number: predict -> run -> inspect -> validate.

Engineering Reflection

Which assumption in this chapter is most unrealistic for a real robot, and why?

What is one line of code you can now explain rather than merely reproduce?

What would you test before using this geometry to control a real robot?

What would you need to add to convert a camera-frame point into a robot-frame point?

Strong answer: A known camera-to-robot extrinsic transformation (rotation and translation), with a clearly defined frame convention.

Chapter takeaway

Camera geometry is the bridge between what a camera measures and where something may be in the physical world. The deeper lesson is engineering: make the model explicit, keep functions modular, test edge cases, visualise intermediate results, and validate the result rather than assuming that successful execution means correctness.

Appendix A - Suggested Student Workflow

1. Open a new notebook or Python file and import NumPy, OpenCV and Matplotlib.
2. Write down the coordinate conventions before writing equations.
3. Define the camera matrix K and image dimensions.
4. Manually calculate the first projection for (0.5, 0.2, 2.0).
5. Run the minimal projection code and compare with your manual result.
6. Turn the projection equations into a reusable function.
7. Add the $Z \leq 0$ guard.
8. Add a separate visibility test.
9. Run all five workshop test points.
10. Predict where the visible points should appear before drawing them.
11. Visualise the points and inspect the image.
12. Back-project (470,300) at depth 2.0 m.
13. Reproject the recovered point.
14. Use `np.allclose()` to validate the round trip.
15. Change one parameter at a time and record what you learn.
16. Only then consider extending the system to more realistic robotics tasks.

Appendix B - Key Functions and Packages

Package / function	Use in this chapter	What to remember
NumPy / <code>np.array</code>	Represent K and 3D points.	Most geometry is array-based.
<code>np.asarray</code>	Normalise input to a NumPy array.	Useful for reusable functions.
K @ point	Matrix-vector camera projection.	Compact camera model representation.
<code>np.round</code>	Round before drawing.	Do not round the geometry too early.
<code>np.allclose</code>	Numerical validation.	Allows small floating-point differences.
OpenCV / <code>cv2.circle</code>	Draw projected points.	Drawing coordinates are discrete.
OpenCV / <code>cv2.putText</code>	Label points.	Useful for visual debugging.
Matplotlib / <code>plt.imshow</code>	Display the generated image.	Convert BGR->RGB if displaying OpenCV images.
<code>cv2.imwrite</code>	Save the visualisation.	Useful for reports and debugging records.

Appendix C - Expected Numerical Results

These values are useful as a debugging reference for the supplied workshop test points.

```

P0 = (0.0, 0.0, 2.0) -> (320.0, 240.0)
P1 = (0.5, 0.2, 2.0) -> (470.0, 300.0)
P2 = (-0.4, 0.1, 1.5) -> (160.0, 280.0)
P3 = (0.3, -0.2, 3.0) -> (380.0, 200.0)
P4 = (0.0, 0.0, -1.0) -> rejected / BEHIND CAMERA

```

```
Back projection:  
pixel = (470, 300), depth = 2.0  
-> (X,Y,Z) = (0.5, 0.2, 2.0)  
-> reprojected pixel = (470, 300)  
-> np.allclose(...) = True
```