Restarted myenv (Python 3.10.6)

In [ ]:
```
#
# LIGHTWEIGHT VISUAL DIFFERENCE BENCHMARKING HARNESS
#
# This script is designed to run on *small samples* of datasets.
# It now includes VISUAL OUTPUTS for each benchmark.
#
# 1. Install dependencies in Cell 0.
# 2. Download datasets as per the "Lightweight Download Commands".
# 3. Create 'my_manual_images/' folder and add your 3 lighting photos.
# 4. **IMPORTANT:** Create one image 'my_clean_image.png' in 'my_manual_images/'
#    (You can copy one from 'datasets/mvtec_ad/carpet/train/good/000.png').
# 5. Run each cell block (separated by #%%) one by one.
#
```

In [ ]:
```
#
# CELL 0: DEPENDENCIES
#
print("--- [CELL 0] Installing Dependencies ---")
import subprocess
import sys

def install(package):
    subprocess.check_call([sys.executable, "-m", "pip", "install", package])

try:
    install("opencv-python")
    install("numpy")
    install("matplotlib")
    install("scikit-image")
    install("scikit-learn")
    # install("opencv-python")  # Already included
    print("✅ All packages are installed.")
except Exception as e:
    print(f"❌ Error installing packages. Please install manually: {e}")
```

```
--- [CELL 0] Installing Dependencies ---
✅ All packages are installed.
```

In [ ]:
```
#
# CELL 1: MANDATORY SETUP (Standardization & Helpers)
#
print("\n--- [CELL 1] Loading Libraries & Helper Functions ---")
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
from sklearn.metrics import f1_score, jaccard_score
from pathlib import Path
import os
import time
```

```
# --- Mandatory Standardization Function ---
TARGET_SIZE = (1024, 1024)

def standardize_image(image_path):
    """Loads, resizes, and normalizes an image."""
    img = cv2.imread(str(image_path))
    if img is None:
        print(f"Error: Could not load image at {image_path}")
        return None

    # 1. Resize
    img_resized = cv2.resize(img, TARGET_SIZE, interpolation=cv2.INTER_AREA)

    # 2. Normalize Bit Depth (to 0.0 - 1.0 float)
    img_normalized = img_resized.astype(np.float32) / 255.0

    return img_normalized

# --- Helper Function for Visualization ---
def show_image(image, title='Image', cmap=None, clear=True):
    """Displays an image using matplotlib."""
    if clear:
        plt.close('all')
    plt.figure(figsize=(6, 6))
    plt.imshow(image, cmap=cmap)
    plt.title(title)
    plt.axis('off')
    plt.show()

print("✅ Setup cell complete. `standardize_image` function is ready.")
```

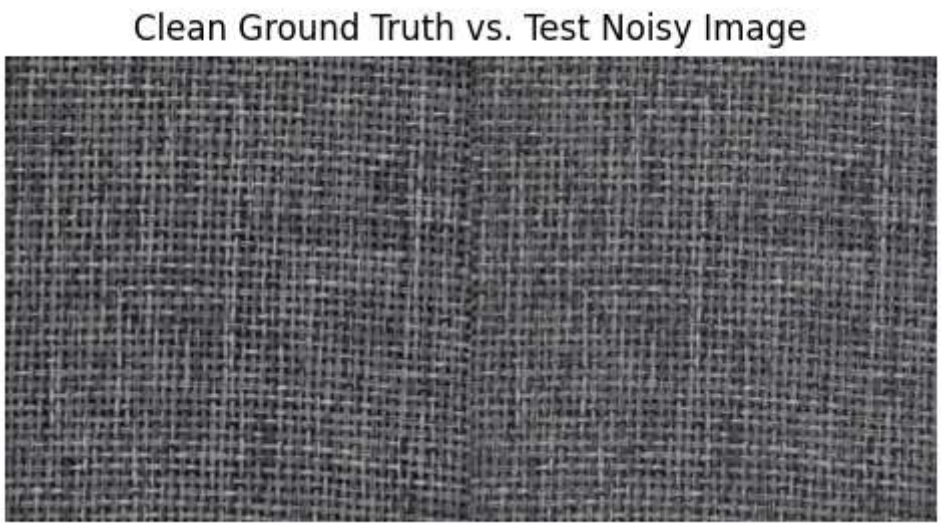
```
--- [CELL 1] Loading Libraries & Helper Functions ---
✅ Setup cell complete. `standardize_image` function is ready.
```

In [ ]:

```
#
# CELL 2.1: NOISE - SETUP (Create Test Set)
#
print("\n--- [CELL 2.1] Noise Benchmark - Setup ---")

# --- 1. Provide a path to a clean, high-quality image ---
#    (You can copy one from the MVTec 'carpet' folder you downloaded)
YOUR_CLEAN_IMAGE_PATH = 'datasets/my_manual_images/my_clean_image.png' # Note: typo

# Create a 'test_data' directory
Path('test_data').mkdir(exist_ok=True)
CLEAN_IMG_PATH = 'test_data/clean_ground_truth.png'
NOISY_IMG_PATH = 'test_data/test_noisy.png'

def add_noise(image):
    """Adds a mix of Gaussian and salt-and-pepper noise."""
    img_float = image.astype(np.float32)

    # Gaussian Noise
    mean = 0
    var = 0.01
    sigma = var**0.5
```

```
    gaussian = np.random.normal(mean, sigma, img_float.shape).astype(np.float32)
    img_with_gaussian = img_float + gaussian

    # Salt & Pepper Noise
    s_vs_p = 0.5
    amount = 0.04
    img_with_noise = img_with_gaussian.copy()

    # Salt
    num_salt = np.ceil(amount * img_with_noise.size * s_vs_p)
    coords = [np.random.randint(0, i - 1, int(num_salt)) for i in img_with_noise.sh
    img_with_noise[coords[0], coords[1], :] = 1.0

    # Pepper
    num_pepper = np.ceil(amount* img_with_noise.size * (1. - s_vs_p))
    coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in img_with_noise.
    img_with_noise[coords[0], coords[1], :] = 0.0

    return np.clip(img_with_noise, 0.0, 1.0)

# --- Create the test files ---
if Path(YOUR_CLEAN_IMAGE_PATH).exists():
    # Standardize the clean image and save it
    clean_img = standardize_image(YOUR_CLEAN_IMAGE_PATH)
    cv2.imwrite(CLEAN_IMG_PATH, (clean_img * 255).astype(np.uint8))

    # Create and save the noisy version
    noisy_img = add_noise(clean_img)
    cv2.imwrite(NOISY_IMG_PATH, (noisy_img * 255).astype(np.uint8))

    print(f"✅ Noise test set created:")
    print(f"   - {CLEAN_IMG_PATH}")
    print(f"   - {NOISY_IMG_PATH}")
    show_image(np.hstack([clean_img, noisy_img]), "Clean Ground Truth vs. Test Nois
else:
    print(f"❌ Error: Path not found. Please set `YOUR_CLEAN_IMAGE_PATH` to a valic
    print("   (e.g., 'my_manual_images/my_clean_image.png')")
```

```
--- [CELL 2.1] Noise Benchmark - Setup ---
✅ Noise test set created:
   - test_data/clean_ground_truth.png
   - test_data/test_noisy.png
```



In [ ]:
```
#
# CELL 2.2: NOISE - HARNESS (Run the Benchmark)
#
print("\n--- [CELL 2.2] Noise Benchmark - Harness ---")

# --- Define Noise Reduction Approaches ---
def no_filter(img):
    return img

def gaussian_filter(img):
    # Convert back to 8-bit for OpenCV filter, then re-normalize
    img_8bit = (img * 255).astype(np.uint8)
    filtered = cv2.GaussianBlur(img_8bit, (5, 5), 0)
    return filtered.astype(np.float32) / 255.0

def median_filter(img):
    img_8bit = (img * 255).astype(np.uint8)
    filtered = cv2.medianBlur(img_8bit, 5)
    return filtered.astype(np.float32) / 255.0

def bilateral_filter(img):
    img_8bit = (img * 255).astype(np.uint8)
    filtered = cv2.bilateralFilter(img_8bit, d=9, sigmaColor=75, sigmaSpace=75)
    return filtered.astype(np.float32) / 255.0

# --- Benchmark Setup ---
noise_approaches = [
    {"name": "No Filter (Baseline)", "func": no_filter},
    {"name": "Gaussian Blur", "func": gaussian_filter},
    {"name": "Median Blur", "func": median_filter},
    {"name": "Bilateral Filter", "func": bilateral_filter},
]

benchmark_results = []

# --- Load Test Data ---
clean_img = standardize_image(CLEAN_IMG_PATH)
noisy_img = standardize_image(NOISY_IMG_PATH)
```

```
if clean_img is not None:
    print("--- 🌫️ Running Noise Reduction Benchmark ---")

    approach_outputs = {} # Store outputs for visualization

    for approach in noise_approaches:
        print(f"Testing: {approach['name']}...")
        start_time = time.time()

        # Apply the filter
        filtered_image = approach['func'](noisy_img)
        approach_outputs[approach['name']] = filtered_image

        duration = time.time() - start_time

        # Calculate PSNR against the *original clean image*
        metric_psnr = psnr(clean_img, filtered_image, data_range=1.0)

        benchmark_results.append({
            "name": approach['name'],
            "psnr": metric_psnr,
            "time": duration
        })

    # --- Print Results ---
    print("\n--- 🏁 Noise Benchmark Results ---")
    results_sorted = sorted(benchmark_results, key=lambda x: x['psnr'], reverse=Tru
    for res in results_sorted:
        print(f"  {res['name']:<25} | PSNR: {res['psnr']:.2f} (Higher is better) |

    winner_name = results_sorted[0]['name']
    print(f"\n🏆 Winner: {winner_name}")
    print("Interpretation: The method with the **highest PSNR** wins. It removed th

    # --- NEW: Add Visual Output ---
    print("\n--- 👁️ Noise Visual Output ---")
    winner_output = approach_outputs[winner_name]
    show_image(np.hstack([noisy_img, winner_output]), f"Noisy Input vs. Winner ({wi

else:
    print("❌ Cannot run harness. Please fix the setup cell first.")
```

```
--- [CELL 2.2] Noise Benchmark - Harness ---
--- 🌫️ Running Noise Reduction Benchmark ---
Testing: No Filter (Baseline)...
Testing: Gaussian Blur...
Testing: Median Blur...
Testing: Bilateral Filter...

--- 🏁 Noise Benchmark Results ---
  Median Blur               | PSNR: 25.47 (Higher is better) | Time: 0.0344s
  Gaussian Blur             | PSNR: 23.23 (Higher is better) | Time: 0.0763s
  Bilateral Filter          | PSNR: 14.77 (Higher is better) | Time: 0.0614s
  No Filter (Baseline)      | PSNR: 13.84 (Higher is better) | Time: 0.0000s

🏆 Winner: Median Blur
Interpretation: The method with the **highest PSNR** wins. It removed the most noise
while staying closest to the original, clean image.

--- 👁️ Noise Visual Output ---
```

In [ ]:

```python
#
# CELL 3.1: LIGHTING - SETUP (Create Test Set)
#
print("\n--- [CELL 3.1] Lighting Benchmark - Setup ---")

# --- 1. Provide paths to your 3 self-created lighting images ---
#    (You MUST create these images yourself and save them)
YOUR_LIGHT_NORMAL_PATH = 'datasets/my_manual_images/my_light_normal.png'
YOUR_LIGHT_BRIGHT_PATH = 'datasets/my_manual_images/my_light_bright.png'
YOUR_LIGHT_SHADOW_PATH = 'datasets/my_manual_images/my_light_shadow.png'

# Create a 'test_data' directory if it doesn't exist
Path('test_data').mkdir(exist_ok=True)
LIGHT_NORMAL_STD_PATH = 'test_data/light_normal_std.png'
LIGHT_BRIGHT_STD_PATH = 'test_data/light_bright_std.png'
LIGHT_SHADOW_STD_PATH = 'test_data/light_shadow_std.png'

# --- Create the test files ---
paths = [YOUR_LIGHT_NORMAL_PATH, YOUR_LIGHT_BRIGHT_PATH, YOUR_LIGHT_SHADOW_PATH]
out_paths = [LIGHT_NORMAL_STD_PATH, LIGHT_BRIGHT_STD_PATH, LIGHT_SHADOW_STD_PATH]
all_exist = all(Path(p).exists() for p in paths)
```

```
if all_exist:
    std_images = []
    for in_path, out_path in zip(paths, out_paths):
        std_img = standardize_image(in_path)
        if std_img is not None:
            std_images.append(std_img)
            cv2.imwrite(out_path, (std_img * 255).astype(np.uint8))
        else:
            all_exist = False
            break

    if all_exist:
        print("✅ Lighting test set standardized and saved.")
        show_image(np.hstack(std_images), "Normal vs. Bright vs. Shadow (Standardiz
    else:
        print("❌ Error: One or more images failed to load.")

else:
    print("❌ Error: One or more paths not found. Please set your 3 lighting image
    print("   (e.g., 'my_manual_images/my_light_normal.jpg')")
```

```
--- [CELL 3.1] Lighting Benchmark - Setup ---
✅ Lighting test set standardized and saved.
```

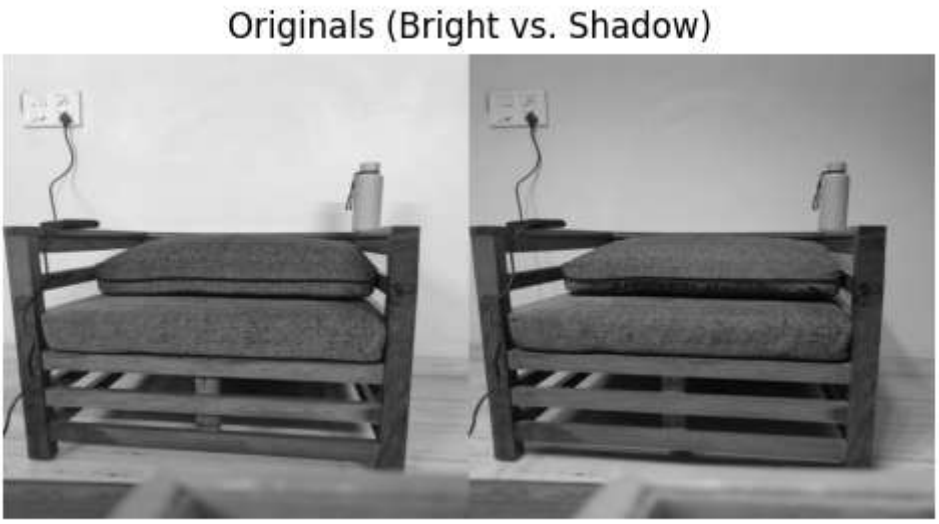
In [ ]:

```
#
# CELL 3.2: LIGHTING - HARNESS (Run the Benchmark)
#
print("\n--- [CELL 3.2] Lighting Benchmark - Harness ---")

# --- Define Lighting Normalization Approaches ---
def no_normalization(img_gray):
    return img_gray

def global_hist_equalize(img_gray):
    # Convert to 8-bit, equalize, re-normalize
    img_8bit = np.clip(img_gray * 255, 0, 255).astype(np.uint8)
    equalized = cv2.equalizeHist(img_8bit)
    return equalized.astype(np.float32) / 255.0

def clahe_normalization(img_gray):
    img_8bit = np.clip(img_gray * 255, 0, 255).astype(np.uint8)
    # Create a CLAHE object (best to define it once)
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    normalized = clahe.apply(img_8bit)
```

```
    return normalized.astype(np.float32) / 255.0

# --- Benchmark Setup ---
normalization_approaches = [
    {"name": "No Normalization (Baseline)", "func": no_normalization},
    {"name": "Global Hist. Equalization", "func": global_hist_equalize},
    {"name": "CLAHE (Adaptive)", "func": clahe_normalization},
]

benchmark_results = []
approach_outputs = {} # Store outputs for visualization

# --- Load Test Data ---
bright_img_std = standardize_image(LIGHT_BRIGHT_STD_PATH)
shadow_img_std = standardize_image(LIGHT_SHADOW_STD_PATH)

if bright_img_std is not None and shadow_img_std is not None:
    print("--- 💡 Running Lighting & Contrast Benchmark ---")

    # Convert to grayscale for normalization
    bright_gray = cv2.cvtColor(bright_img_std, cv2.COLOR_BGR2GRAY)
    shadow_gray = cv2.cvtColor(shadow_img_std, cv2.COLOR_BGR2GRAY)

    for approach in normalization_approaches:
        print(f"Testing: {approach['name']}...")
        start_time = time.time()

        # Apply the method to BOTH images
        norm_bright = approach['func'](bright_gray)
        norm_shadow = approach['func'](shadow_gray)

        approach_outputs[approach['name']] = (norm_bright, norm_shadow)

        duration = time.time() - start_time

        # Calculate SSIM between the two processed images
        metric_ssim = ssim(norm_bright, norm_shadow, data_range=1.0)

        benchmark_results.append({
            "name": approach['name'],
            "ssim": metric_ssim,
            "time": duration
        })

    # --- Print Results ---
    print("\n--- 🏁 Lighting Benchmark Results ---")
    results_sorted = sorted(benchmark_results, key=lambda x: x['ssim'], reverse=Tru
    for res in results_sorted:
        print(f"  {res['name']:<30} | SSIM: {res['ssim']:.4f} (Higher is better) |

    winner_name = results_sorted[0]['name']
    print(f"\n🏆 Winner: {winner_name}")
    print("Interpretation: The method with the **highest SSIM** wins. It made the t

    # --- NEW: Add Visual Output ---
    print("\n--- 👁️ Lighting Visual Output ---")
```

```
    # Get the original gray images
    orig_bright_gray = cv2.cvtColor(bright_img_std, cv2.COLOR_BGR2GRAY)
    orig_shadow_gray = cv2.cvtColor(shadow_img_std, cv2.COLOR_BGR2GRAY)
    # Get the winner's processed images
    winner_bright, winner_shadow = approach_outputs[winner_name]

    show_image(np.hstack([orig_bright_gray, orig_shadow_gray]), "Originals (Bright
    show_image(np.hstack([winner_bright, winner_shadow]), f"Winner ({winner_name})

else:
    print("❌ Cannot run harness. Please fix the setup cell first.")
```

```
--- [CELL 3.2] Lighting Benchmark - Harness ---
--- 💡 Running Lighting & Contrast Benchmark ---
Testing: No Normalization (Baseline)...
Testing: Global Hist. Equalization...
Testing: CLAHE (Adaptive)...

--- 🏁 Lighting Benchmark Results ---
  No Normalization (Baseline)    | SSIM: 0.7796 (Higher is better) | Time: 0.0000s
  CLAHE (Adaptive)               | SSIM: 0.7065 (Higher is better) | Time: 0.0163s
  Global Hist. Equalization      | SSIM: 0.6519 (Higher is better) | Time: 0.0169s

🏆 Winner: No Normalization (Baseline)
Interpretation: The method with the **highest SSIM** wins. It made the two different
ly-lit images look the most similar.

--- 👁️ Lighting Visual Output ---
```



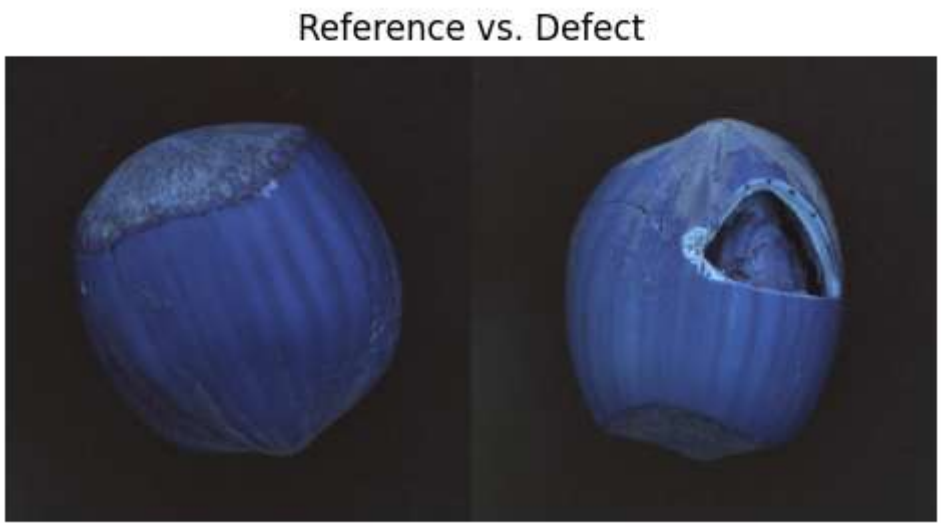
Winner (No Normalization (Baseline)) Output

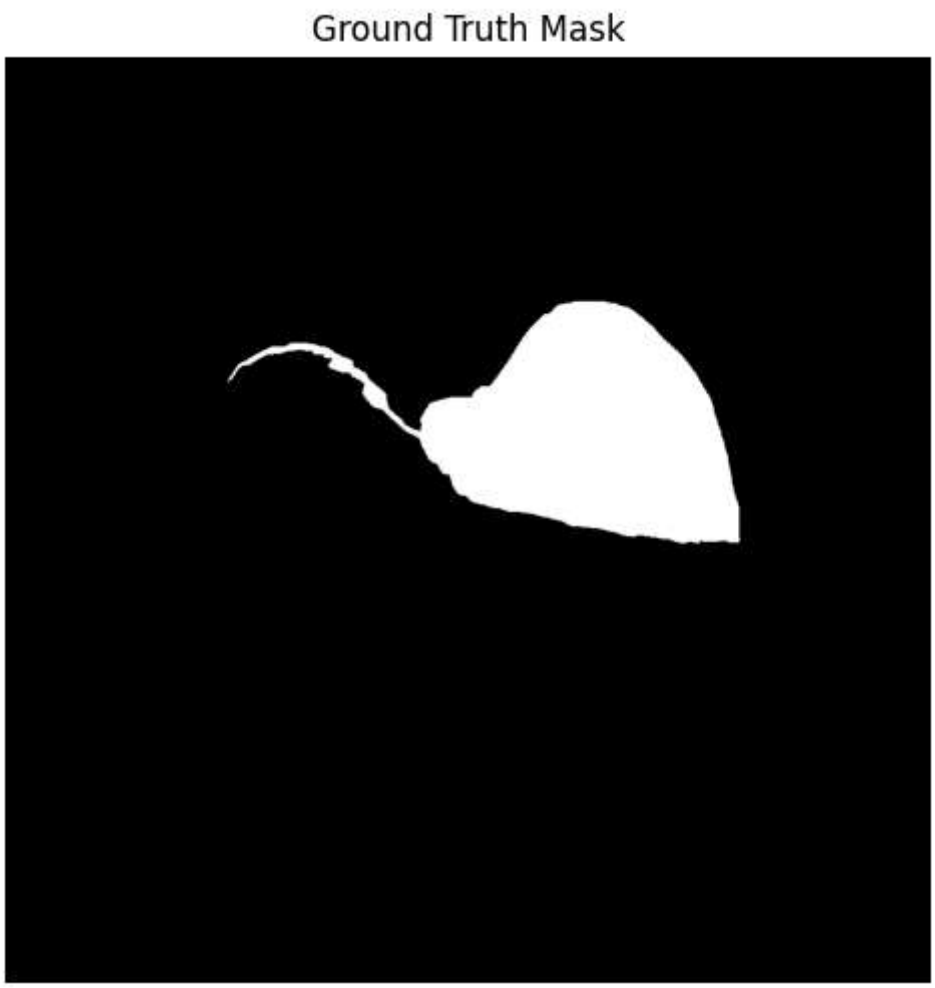
In [ ]:

```
#
# CELL 4.1: COLOR SPACE - SETUP (Get Test Set)
#
print("\n--- [CELL 4.1] Color Space Benchmark - Setup ---")

# --- 1. Provide paths to a defect test case (Ref, Defect, Mask) ---
#    (Using MVTec 'hazelnut' for this color-based test)
YOUR_REF_PATH = 'datasets/mvtec_ad/hazelnut/train/good/000.png'
YOUR_DEFECT_PATH = 'datasets/mvtec_ad/hazelnut/test/crack/000.png'
YOUR_MASK_PATH = 'datasets/mvtec_ad/hazelnut/ground_truth/crack/000_mask.png'

# Create a 'test_data' directory
Path('test_data').mkdir(exist_ok=True)
COLOR_REF_PATH = 'test_data/color_ref.png'
COLOR_DEFECT_PATH = 'test_data/color_defect.png'
COLOR_MASK_PATH = 'test_data/color_mask.png'

# --- Create the test files ---
paths = [YOUR_REF_PATH, YOUR_DEFECT_PATH, YOUR_MASK_PATH]
out_paths = [COLOR_REF_PATH, COLOR_DEFECT_PATH, COLOR_MASK_PATH]
all_exist = all(Path(p).exists() for p in paths)

if all_exist:
    # Standardize all three images
    ref_img = standardize_image(YOUR_REF_PATH)
    defect_img = standardize_image(YOUR_DEFECT_PATH)

    # Load mask and standardize (it's grayscale)
    mask = cv2.imread(YOUR_MASK_PATH, cv2.IMREAD_GRAYSCALE)
    mask_resized = cv2.resize(mask, TARGET_SIZE, interpolation=cv2.INTER_NEAREST)
    # Binarize the mask (must be 0 or 1)
    mask_binary = (mask_resized > 0).astype(np.uint8)

    # Save them
    if ref_img is not None and defect_img is not None:
        cv2.imwrite(COLOR_REF_PATH, (ref_img * 255).astype(np.uint8))
        cv2.imwrite(COLOR_DEFECT_PATH, (defect_img * 255).astype(np.uint8))
        cv2.imwrite(COLOR_MASK_PATH, mask_binary * 255)
```

```
        print("✅ Color Space test set standardized and saved.")
        show_image(np.hstack([ref_img, defect_img]), "Reference vs. Defect")
        show_image(mask_binary, "Ground Truth Mask", cmap='gray')
    else:
        print("❌ Error: Failed to load reference or defect image.")
else:
    print("❌ Error: One or more paths not found. Please set your 3 defect image pa
    print("   (e.g., 'datasets/mvtec_ad/hazelnut/train/good/000.png')")
```

```
--- [CELL 4.1] Color Space Benchmark - Setup ---
✅ Color Space test set standardized and saved.
```



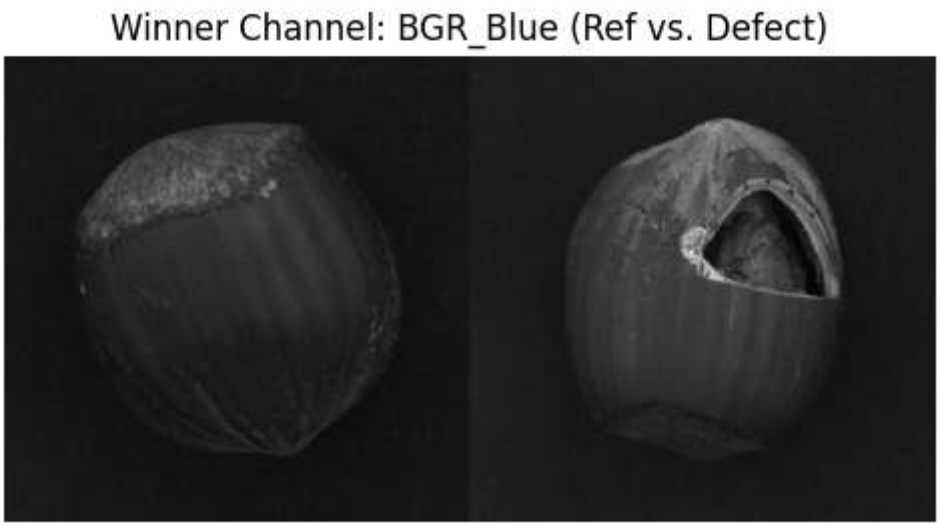


In [ ]:
```
#
# CELL 4.2: COLOR SPACE - HARNESS (Run the Benchmark)
#
print("\n--- [CELL 4.2] Color Space Benchmark - Harness ---")

def get_all_channels(img_float_bgr):
    """Extracts all relevant single channels for comparison."""
    # BGR
    b, g, r = cv2.split(img_float_bgr)

    # Grayscale
    gray = cv2.cvtColor(img_float_bgr, cv2.COLOR_BGR2GRAY)

    # HSV
    hsv = cv2.cvtColor(img_float_bgr, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)

    # L*a*b*
    lab = cv2.cvtColor(img_float_bgr, cv2.COLOR_BGR2Lab)
    l, a, b_lab = cv2.split(lab)

    return {
        "BGR_Blue": b,
        "BGR_Green": g,
        "BGR_Red": r,
```

```
        "Grayscale": gray,
        "HSV_Hue": h,
        "HSV_Saturation": s,
        "HSV_Value": v,
        "LAB_Lightness": l,
        "LAB_A (Green-Red)": a,
        "LAB_B (Blue-Yellow)": b_lab,
    }

# --- Benchmark Setup ---
benchmark_results = []
approach_outputs = {} # Store outputs for visualization

# --- Load Test Data ---
ref_img = standardize_image(COLOR_REF_PATH)
defect_img = standardize_image(COLOR_DEFECT_PATH)
mask = cv2.imread(COLOR_MASK_PATH, cv2.IMREAD_GRAYSCALE)

if ref_img is not None and defect_img is not None and mask is not None:
    # Flatten mask for scikit-learn metrics
    mask_flat = (mask > 0).flatten()
    print("--- 🎨 Running Color Space Benchmark ---")

    # Get all channels for both images
    ref_channels = get_all_channels(ref_img)
    defect_channels = get_all_channels(defect_img)

    for channel_name, ref_chan in ref_channels.items():
        print(f"Testing: {channel_name}...")
        defect_chan = defect_channels[channel_name]

        # --- Simple Detector: Absolute Difference + Threshold ---
        diff = cv2.absdiff(ref_chan, defect_chan)

        # Use Otsu's threshold to automatically find the "change"
        # We must convert 0-1 float to 0-255 uint8 for Otsu
        diff_8bit = np.clip(diff * 255, 0, 255).astype(np.uint8)
        _, detected_mask = cv2.threshold(diff_8bit, 0, 255, cv2.THRESH_BINARY + cv2

        # --- Store outputs for viz ---
        approach_outputs[channel_name] = {
            "ref_chan": ref_chan,
            "defect_chan": defect_chan,
            "detected_mask": (detected_mask > 0).astype(np.uint8)
        }

        # --- Metric: F1-Score ---
        detected_flat = (detected_mask > 0).flatten()
        metric_f1 = f1_score(mask_flat, detected_flat, zero_division=0)

        benchmark_results.append({
            "name": channel_name,
            "f1_score": metric_f1,
        })

    # --- Print Results ---
```

```
    print("\n--- 🏁 Color Space Benchmark Results ---")
    results_sorted = sorted(benchmark_results, key=lambda x: x['f1_score'], reverse
    for res in results_sorted:
        print(f"  {res['name']:<25} | F1-Score: {res['f1_score']:.4f} (Higher is be

    winner_name = results_sorted[0]['name']
    print(f"\n🏆 Winner: {winner_name}")
    print(f"Interpretation: The channel with the **highest F1-Score** is the best f

    # --- NEW: Add Visual Output ---
    print("\n--- 👁️ Color Space Visual Output ---")
    winner_data = approach_outputs[winner_name]

    # Show the winning channel
    show_image(np.hstack([winner_data['ref_chan'], winner_data['defect_chan']]),
               f"Winner Channel: {winner_name} (Ref vs. Defect)", cmap='gray')

    # Show the resulting mask
    show_image(np.hstack([winner_data['detected_mask'], mask_flat.reshape(TARGET_SI
               "Detected Mask (Winner) vs. Ground Truth Mask", cmap='gray')

else:
    print("❌ Cannot run harness. Please fix the setup cell first.")
```

```
--- [CELL 4.2] Color Space Benchmark - Harness ---
--- 🎨 Running Color Space Benchmark ---
Testing: BGR_Blue...
Testing: BGR_Green...
Testing: BGR_Red...
Testing: Grayscale...
Testing: HSV_Hue...
Testing: HSV_Saturation...
Testing: HSV_Value...
Testing: LAB_Lightness...
Testing: LAB_A (Green-Red)...
Testing: LAB_B (Blue-Yellow)...

--- 🏁 Color Space Benchmark Results ---
  BGR_Blue                  | F1-Score: 0.3796 (Higher is better)
  BGR_Green                 | F1-Score: 0.3076 (Higher is better)
  Grayscale                 | F1-Score: 0.3034 (Higher is better)
  HSV_Value                 | F1-Score: 0.2955 (Higher is better)
  BGR_Red                   | F1-Score: 0.2949 (Higher is better)
  HSV_Saturation            | F1-Score: 0.2341 (Higher is better)
  LAB_A (Green-Red)         | F1-Score: 0.1434 (Higher is better)
  LAB_B (Blue-Yellow)       | F1-Score: 0.1340 (Higher is better)
  LAB_Lightness             | F1-Score: 0.1214 (Higher is better)
  HSV_Hue                   | F1-Score: 0.1196 (Higher is better)

🏆 Winner: BGR_Blue
Interpretation: The channel with the **highest F1-Score** is the best for isolating
this *specific type* of defect.

--- 👁️ Color Space Visual Output ---
```



In [ ]:
```
#
# CELL 5.1: BACKGROUND / ROI - SETUP (Get Test Sets)
#
print("\n--- [CELL 5.1] Background/ROI Benchmark - Setup ---")

# --- 1. Provide paths for an OBJECT-centric dataset ---
YOUR_OBJ_REF_PATH = 'datasets/mvtec_ad/bottle/train/good/000.png' # <<< This path i
YOUR_OBJ_DEFECT_PATH = 'datasets/mvtec_ad/bottle/test/broken_small/000.png' # <<< T
YOUR_OBJ_MASK_PATH = 'datasets/mvtec_ad/bottle/ground_truth/broken_small/000_mask.p

# --- 2. Provide paths for a TEXTURE-centric dataset ---
YOUR_TEX_REF_PATH = 'datasets/mvtec_ad/carpet/train/good/000.png' # <<< This path i
YOUR_TEX_DEFECT_PATH = 'datasets/mvtec_ad/carpet/test/hole/000.png' # <<< This path
YOUR_TEX_MASK_PATH = 'datasets/mvtec_ad/carpet/ground_truth/hole/000_mask.png' # <<

# --- Standardize and save all files ---
Path('test_data').mkdir(exist_ok=True)
all_paths = {
    'obj_ref': (YOUR_OBJ_REF_PATH, 'test_data/bg_obj_ref.png'),
    'obj_defect': (YOUR_OBJ_DEFECT_PATH, 'test_data/bg_obj_defect.png'),
    'obj_mask': (YOUR_OBJ_MASK_PATH, 'test_data/bg_obj_mask.png'),
    'tex_ref': (YOUR_TEX_REF_PATH, 'test_data/bg_tex_ref.png'),
```

```
    'tex_defect': (YOUR_TEX_DEFECT_PATH, 'test_data/bg_tex_defect.png'),
    'tex_mask': (YOUR_TEX_MASK_PATH, 'test_data/bg_tex_mask.png'),
}

all_exist = all(Path(p[0]).exists() for p in all_paths.values())

if all_exist:
    for key, (in_path, out_path) in all_paths.items():
        if 'mask' in key:
            mask = cv2.imread(in_path, cv2.IMREAD_GRAYSCALE)
            if mask is None:
                print(f"❌ Error loading mask: {in_path}")
                all_exist = False
                break
            mask_resized = cv2.resize(mask, TARGET_SIZE, interpolation=cv2.INTER_NE
            mask_binary = (mask_resized > 0).astype(np.uint8)
            cv2.imwrite(out_path, mask_binary * 255)
        else:
            std_img = standardize_image(in_path)
            if std_img is None:
                print(f"❌ Error loading image: {in_path}")
                all_exist = False
                break
            cv2.imwrite(out_path, (std_img * 255).astype(np.uint8))
    if all_exist:
        print("✅ Background/ROI test sets standardized and saved.")
    else:
        print("❌ Error during file processing. Check paths.")
else:
    print("❌ Error: One or more paths not found. Please set your 6 dataset paths.'
    print("   (e.g., 'datasets/mvtec_ad/bottle/train/good/000.png')")
```

```
--- [CELL 5.1] Background/ROI Benchmark - Setup ---
✅ Background/ROI test sets standardized and saved.
```

In [ ]:

```
#
# CELL 5.2: BACKGROUND / ROI - HARNESS (Run the Benchmark)
#
print("\n--- [CELL 5.2] Background/ROI Benchmark - Harness ---")

# --- Define Background Handling Approaches ---
print("✅ Using OpenCV GrabCut for background removal (no external dependencies)")

def no_handling(img):
    """Returns the image as-is."""
    return img

def grabcut_remove(img):
    """Uses OpenCV GrabCut to remove background."""
    # Convert to 8-bit for GrabCut
    img_8bit = (img * 255).astype(np.uint8)

    # Create mask
    mask = np.zeros(img_8bit.shape[:2], np.uint8)

    # Background and foreground models
```

```
    bgd_model = np.zeros((1, 65), np.float64)
    fgd_model = np.zeros((1, 65), np.float64)

    # Define rectangle around subject (center 80% of image)
    h, w = img_8bit.shape[:2]
    rect = (int(w*0.1), int(h*0.1), int(w*0.8), int(h*0.8))

    try:
        # Run GrabCut
        cv2.grabCut(img_8bit, mask, rect, bgd_model, fgd_model, 5, cv2.GC_INIT_WITH

        # Create binary mask (0 and 2 are background, 1 and 3 are foreground)
        mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')

        # Apply mask to get foreground
        foreground = img_8bit * mask2[:, :, np.newaxis]

        # Alpha-blend with white background
        alpha = mask2.astype(np.float32)
        alpha = np.expand_dims(alpha, axis=-1)

        bgr = foreground.astype(np.float32) / 255.0
        white_bg = np.ones_like(bgr, dtype=np.float32)
        blended = (bgr * alpha) + (white_bg * (1.0 - alpha))

        return blended
    except Exception as e:
        print(f"⚠️ GrabCut failed: {e}, returning original")
        return img

def static_roi_mask(img):
    """Applies a hard-coded circular ROI. (Good for bottles)."""
    mask = np.zeros(img.shape[:2], dtype=np.uint8)
    # Example: A circular ROI in the center
    center = (TARGET_SIZE[0] // 2, TARGET_SIZE[1] // 2)
    radius = int(TARGET_SIZE[0] * 0.4) # 40% of width
    cv2.circle(mask, center, radius, 255, -1)

    # Apply mask
    img_masked = cv2.bitwise_and(img, img, mask=mask)
    # Add 1.0 (white) where mask was 0, to avoid black background
    img_masked[mask == 0] = 1.0
    return img_masked

# --- Benchmark Setup ---
bg_approaches = [
    {"name": "No Handling (Baseline)", "func": no_handling},
    {"name": "GrabCut (BG Removal)", "func": grabcut_remove},
    {"name": "Static ROI Mask", "func": static_roi_mask},
]

test_sets = [
    {
        "name": "Object (Bottle)",
        "ref_path": 'test_data/bg_obj_ref.png',
        "defect_path": 'test_data/bg_obj_defect.png',
```

```
        "mask_path": 'test_data/bg_obj_mask.png'
    },
    {
        "name": "Texture (Carpet)",
        "ref_path": 'test_data/bg_tex_ref.png',
        "defect_path": 'test_data/bg_tex_defect.png',
        "mask_path": 'test_data/bg_tex_mask.png'
    }
]

print("--- 🖼️ Running Background/ROI Benchmark ---")
approach_outputs = {} # Store outputs for visualization

for test_set in test_sets:
    print(f"\n--- Testing Dataset: {test_set['name']} ---")
    benchmark_results = []

    # Load data for this set
    ref_img = standardize_image(test_set['ref_path'])
    defect_img = standardize_image(test_set['defect_path'])
    mask = cv2.imread(test_set['mask_path'], cv2.IMREAD_GRAYSCALE)

    if ref_img is None or defect_img is None or mask is None:
        print("❌ Skipping, test data not loaded. Did you run the setup cell?")
        continue

    mask_flat = (mask > 0).flatten()
    approach_outputs[test_set['name']] = {}

    for approach in bg_approaches:
        print(f"  Testing: {approach['name']}...")
        start_time = time.time()

        # Apply approach to both images
        ref_processed = approach['func'](ref_img)
        defect_processed = approach['func'](defect_img)

        approach_outputs[test_set['name']][approach['name']] = (ref_processed, defe

        duration = time.time() - start_time

        # --- Simple Detector: SSIM ---
        # We need grayscale for SSIM
        ref_gray = cv2.cvtColor(ref_processed, cv2.COLOR_BGR2GRAY)
        defect_gray = cv2.cvtColor(defect_processed, cv2.COLOR_BGR2GRAY)

        # Get the difference map
        (score, diff_map) = ssim(ref_gray, defect_gray, data_range=1.0, full=True)

        # Difference map is 0-1, 0=same, 1=different. We want 1=different.
        diff_map = 1.0 - diff_map

        # Threshold (anything with > 20% diff is a change)
        detected_mask = (diff_map > 0.2)

        # --- Metric: F1-Score ---
```

```
        metric_f1 = f1_score(mask_flat, detected_mask.flatten(), zero_division=0)

        benchmark_results.append({
            "name": approach['name'],
            "f1_score": metric_f1,
            "time": duration
        })

    # --- Print Results ---
    results_sorted = sorted(benchmark_results, key=lambda x: x['f1_score'], reverse
    for res in results_sorted:
        print(f"    {res['name']:<25} | F1-Score: {res['f1_score']:.4f} (Higher is

    winner_name = results_sorted[0]['name']
    print(f"  🏆 Winner for {test_set['name']}: {winner_name}")

    # --- NEW: Add Visual Output ---
    # Show the "Object" test set results, as they are more visual
    if test_set['name'] == "Object (Bottle)":
        print("\n--- 👁️ Background Visual Output (Object) ---")

        # Show the "No Handling" output
        no_handle_ref, _ = approach_outputs[test_set['name']]["No Handling (Baselin
        show_image(no_handle_ref, "Original Image (Baseline)")

        # Show the "GrabCut" output
        grabcut_ref, _ = approach_outputs[test_set['name']]["GrabCut (BG Removal)"]
        show_image(grabcut_ref, "GrabCut Output")

        # Show the "ROI" output
        roi_ref, _ = approach_outputs[test_set['name']]["Static ROI Mask"]
        show_image(roi_ref, "Static ROI Output")

print("\nInterpretation: This test proves that the 'best' method is **context-depen
print("  - 'GrabCut' or 'ROI Mask' should win for 'Object'.")
print("  - 'No Handling' should win for 'Texture'.")
```

```
--- [CELL 5.2] Background/ROI Benchmark - Harness ---
✅ Using OpenCV GrabCut for background removal (no external dependencies)
--- 🖼️ Running Background/ROI Benchmark ---

--- Testing Dataset: Object (Bottle) ---
  Testing: No Handling (Baseline)...
  Testing: GrabCut (BG Removal)...
  Testing: Static ROI Mask...
    Static ROI Mask           | F1-Score: 0.0635 (Higher is better) | Time: 0.1598s
    No Handling (Baseline)    | F1-Score: 0.0551 (Higher is better) | Time: 0.0000s
    GrabCut (BG Removal)      | F1-Score: 0.0543 (Higher is better) | Time: 13.2537s
  🏆 Winner for Object (Bottle): Static ROI Mask

--- 👁️ Background Visual Output (Object) ---
```

Original Image (Baseline)

GrabCut Output

Static ROI Output

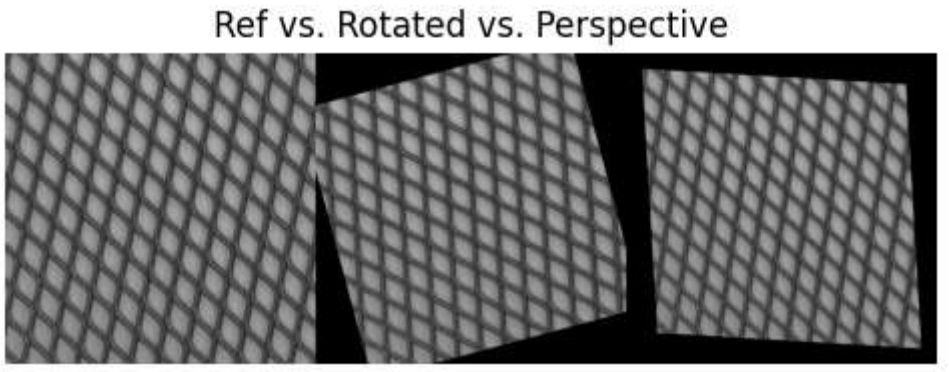
```
--- Testing Dataset: Texture (Carpet) ---
  Testing: No Handling (Baseline)...
  Testing: GrabCut (BG Removal)...
  Testing: Static ROI Mask...
    Static ROI Mask           | F1-Score: 0.0467 (Higher is better) | Time: 0.0811s
    No Handling (Baseline)    | F1-Score: 0.0234 (Higher is better) | Time: 0.0000s
    GrabCut (BG Removal)      | F1-Score: 0.0000 (Higher is better) | Time: 17.0604s
  🏆 Winner for Texture (Carpet): Static ROI Mask

Interpretation: This test proves that the 'best' method is **context-dependent**.
  - 'GrabCut' or 'ROI Mask' should win for 'Object'.
  - 'No Handling' should win for 'Texture'.
```

In [ ]:

```python
#
# CELL 6.1: ALIGNMENT - SETUP (Create Transform Set)
#
print("\n--- [CELL 6.1] Alignment Benchmark - Setup ---")

# --- 1. Provide a path to a "good" image, preferably with a grid ---
#    (MVTec 'grid' 'train/good' is perfect for this)
YOUR_GRID_REF_PATH = 'datasets/mvtec_ad/grid/train/good/000.png' # <<< This path is

# Create a 'test_data' directory
Path('test_data').mkdir(exist_ok=True)
ALIGN_REF_PATH = 'test_data/align_ref.png'
ALIGN_ROTATED_PATH = 'test_data/align_rotated.png'
```

```
ALIGN_PERSPECTIVE_PATH = 'test_data/align_perspective.png'

if Path(YOUR_GRID_REF_PATH).exists():
    # Standardize the reference image
    ref_img = standardize_image(YOUR_GRID_REF_PATH)
    if ref_img is not None:
        cv2.imwrite(ALIGN_REF_PATH, (ref_img * 255).astype(np.uint8))
        h, w = TARGET_SIZE

        # --- 1. Create Rotated Image ---
        angle = 15
        scale = 0.9
        M_rot = cv2.getRotationMatrix2D((w/2, h/2), angle, scale)
        img_rotated = cv2.warpAffine(ref_img, M_rot, (w, h))
        cv2.imwrite(ALIGN_ROTATED_PATH, (img_rotated * 255).astype(np.uint8))

        # --- 2. Create Perspective Warped Image ---
        pts1 = np.float32([[0, 0], [w, 0], [0, h], [w, h]])
        pts2 = np.float32([[w*0.05, h*0.05], [w*0.9, h*0.1], [w*0.1, h*0.9], [w*0.9
        M_pers = cv2.getPerspectiveTransform(pts1, pts2)
        img_perspective = cv2.warpPerspective(ref_img, M_pers, (w, h))
        cv2.imwrite(ALIGN_PERSPECTIVE_PATH, (img_perspective * 255).astype(np.uint8

        print("✅ Alignment test set standardized and saved.")
        show_image(np.hstack([ref_img, img_rotated, img_perspective]), "Ref vs. Rot
    else:
        print(f"❌ Error: Failed to load image at {YOUR_GRID_REF_PATH}")
else:
    print(f"❌ Error: Path not found. Please set `YOUR_GRID_REF_PATH`.")
    print("   (e.g., 'datasets/mvtec_ad/grid/train/good/000.png')")
```

```
--- [CELL 6.1] Alignment Benchmark - Setup ---
✅ Alignment test set standardized and saved.
```



In [ ]:

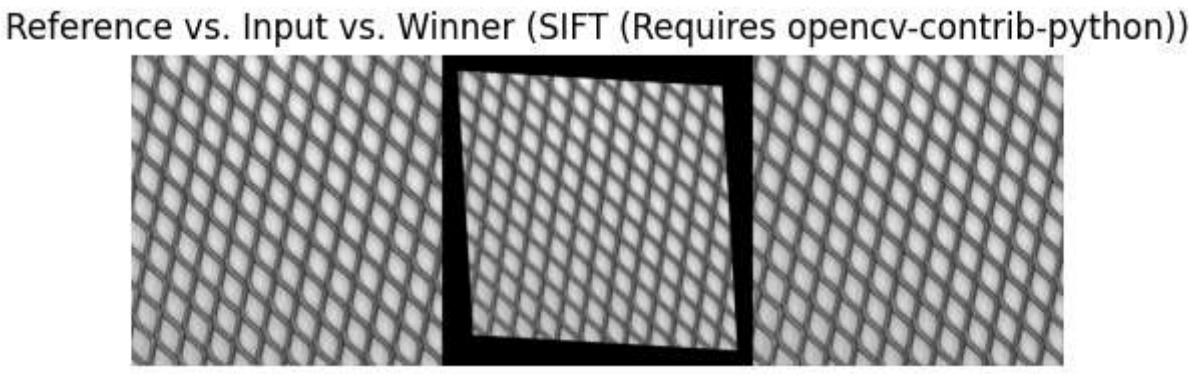
```
#
# CELL 6.2: ALIGNMENT - HARNESS (Run the Benchmark)
#
print("\n--- [CELL 6.2] Alignment Benchmark - Harness ---")

# --- Define Alignment Approaches ---
# These must work on 0-1 float, grayscale images
def align_orb(img1_gray, img2_gray):
    """Aligns img2 to img1 using ORB."""
    # ORB needs 8-bit images
    img1_8bit = (img1_gray * 255).astype(np.uint8)
```

```
    img2_8bit = (img2_gray * 255).astype(np.uint8)
    h, w = img1_8bit.shape

    orb = cv2.ORB_create(nfeatures=5000)
    kp1, des1 = orb.detectAndCompute(img1_8bit, None)
    kp2, des2 = orb.detectAndCompute(img2_8bit, None)

    if des1 is None or des2 is None or len(kp1) < 10 or len(kp2) < 10:
        return img2_gray # Failed

    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)

    if len(matches) < 10: return img2_gray # Failed

    pts1 = np.float32([kp1[m.queryIdx].pt for m in matches[:50]])
    pts2 = np.float32([kp2[m.trainIdx].pt for m in matches[:50]])

    H, _ = cv2.findHomography(pts2, pts1, cv2.RANSAC, 5.0)
    if H is None: return img2_gray # Failed

    aligned_8bit = cv2.warpPerspective(img2_8bit, H, (w, h))
    return aligned_8bit.astype(np.float32) / 255.0

def align_sift(img1_gray, img2_gray):
    """Aligns img2 to img1 using SIFT."""
    img1_8bit = (img1_gray * 255).astype(np.uint8)
    img2_8bit = (img2_gray * 255).astype(np.uint8)
    h, w = img1_8bit.shape

    try:
        sift = cv2.SIFT_create(nfeatures=5000)
        kp1, des1 = sift.detectAndCompute(img1_8bit, None)
        kp2, des2 = sift.detectAndCompute(img2_8bit, None)
    except cv2.error as e:
        print("SIFT Error: SIFT is in 'opencv-contrib-python'.")
        print("Please run: pip install opencv-contrib-python")
        return img2_gray

    if des1 is None or des2 is None or len(kp1) < 10 or len(kp2) < 10:
        return img2_gray # Failed

    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)

    if len(matches) < 10: return img2_gray # Failed

    pts1 = np.float32([kp1[m.queryIdx].pt for m in matches[:50]])
    pts2 = np.float32([kp2[m.trainIdx].pt for m in matches[:50]])

    H, _ = cv2.findHomography(pts2, pts1, cv2.RANSAC, 5.0)
    if H is None: return img2_gray # Failed

    aligned_8bit = cv2.warpPerspective(img2_8bit, H, (w, h))
```

```
    return aligned_8bit.astype(np.float32) / 255.0

def align_ecc(img1_gray, img2_gray):
    """Aligns img2 to img1 using ECC (intensity-based)."""
    h, w = img1_gray.shape

    # ECC works best with Affine transforms for this
    warp_matrix = np.eye(2, 3, dtype=np.float32)
    criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 5000, 1e-10)

    try:
        (cc, warp_matrix) = cv2.findTransformECC(img1_gray, img2_gray, warp_matrix,
                                                 cv2.MOTION_AFFINE, criteria)
    except cv2.error:
        return img2_gray # Failed

    aligned = cv2.warpAffine(img2_gray, warp_matrix, (w, h),
                             flags=cv2.INTER_LINEAR + cv2.WARP_INVERSE_MAP)
    return aligned

# --- Benchmark Setup ---
alignment_approaches = [
    {"name": "ORB", "func": align_orb},
    {"name":"SIFT (Requires opencv-contrib-python)", "func": align_sift},
    {"name": "ECC (Affine)", "func": align_ecc},
]

test_sets = [
    {"name": "Rotated/Scaled", "path": ALIGN_ROTATED_PATH},
    {"name": "Perspective", "path": ALIGN_PERSPECTIVE_PATH},
]

# --- Load Reference Image ---
ref_img = standardize_image(ALIGN_REF_PATH)

if ref_img is not None:
    ref_gray = cv2.cvtColor(ref_img, cv2.COLOR_BGR2GRAY)
    print("--- 📐 Running Image Alignment Benchmark ---")

    all_outputs = {} # Store all outputs

    for test_set in test_sets:
        print(f"\n--- Testing On: {test_set['name']} ---")
        benchmark_results = []

        test_img = standardize_image(test_set['path'])
        if test_img is None:
            print("❌ Skipping, test data not loaded.")
            continue

        test_gray = cv2.cvtColor(test_img, cv2.COLOR_BGR2GRAY)
        all_outputs[test_set['name']] = {'test_img': test_gray}

        for approach in alignment_approaches:
            print(f"  Testing: {approach['name']}...")
            start_time = time.time()
```

```
            # Apply alignment
            aligned_image = approach['func'](ref_gray, test_gray)
            all_outputs[test_set['name']][approach['name']] = aligned_image

            duration = time.time() - start_time

            # --- Metric: SSIM ---
            # Compare the "fixed" image back to the original reference
            metric_ssim = ssim(ref_gray, aligned_image, data_range=1.0)

            benchmark_results.append({
                "name": approach['name'],
                "ssim": metric_ssim,
                "time": duration
            })

        # --- Print Results ---
        results_sorted = sorted(benchmark_results, key=lambda x: x['ssim'], reverse
        for res in results_sorted:
            print(f"    {res['name']:<35} | SSIM: {res['ssim']:.4f} (Higher is bett

        winner_name = results_sorted[0]['name']
        print(f"  🏆 Winner for {test_set['name']}: {winner_name}")

        # --- NEW: Add Visual Output ---
        # Show the "Perspective" test results
        if test_set['name'] == "Perspective":
            print("\n--- 👁️ Alignment Visual Output (Perspective) ---")
            winner_output = all_outputs['Perspective'][winner_name]
            test_input_img = all_outputs['Perspective']['test_img']
            show_image(np.hstack([ref_gray, test_input_img, winner_output]),
                       f"Reference vs. Input vs. Winner ({winner_name})", cmap='gra

    print("\nInterpretation: The method with the **highest SSIM** wins. It was most
else:
    print("❌ Cannot run harness. Please fix the setup cell first.")

print("\n\n--- ✅ ALL PREPROCESSING BENCHMARKS COMPLETE ---")
```

```
--- [CELL 6.2] Alignment Benchmark - Harness ---
--- 📐 Running Image Alignment Benchmark ---

--- Testing On: Rotated/Scaled ---
  Testing: ORB...
  Testing: SIFT (Requires opencv-contrib-python)...
  Testing: ECC (Affine)...
    SIFT (Requires opencv-contrib-python) | SSIM: 0.9679 (Higher is better) | Time:
0.3709s
    ORB                                 | SSIM: 0.4014 (Higher is better) | Time: 0.
8590s
    ECC (Affine)                        | SSIM: 0.2276 (Higher is better) | Time: 0.
1260s
  🏆 Winner for Rotated/Scaled: SIFT (Requires opencv-contrib-python)

--- Testing On: Perspective ---
  Testing: ORB...
  Testing: SIFT (Requires opencv-contrib-python)...
  Testing: ECC (Affine)...
    SIFT (Requires opencv-contrib-python) | SSIM: 0.9936 (Higher is better) | Time:
0.3031s
    ECC (Affine)                        | SSIM: 0.1963 (Higher is better) | Time: 0.
0463s
    ORB                                 | SSIM: 0.1157 (Higher is better) | Time: 0.
0966s
  🏆 Winner for Perspective: SIFT (Requires opencv-contrib-python)

--- 👁️ Alignment Visual Output (Perspective) ---
```




```
Interpretation: The method with the **highest SSIM** wins. It was most successful at
'un-doing' the transformation and aligning the image back to the reference.


--- ✅ ALL PREPROCESSING BENCHMARKS COMPLETE ---
```