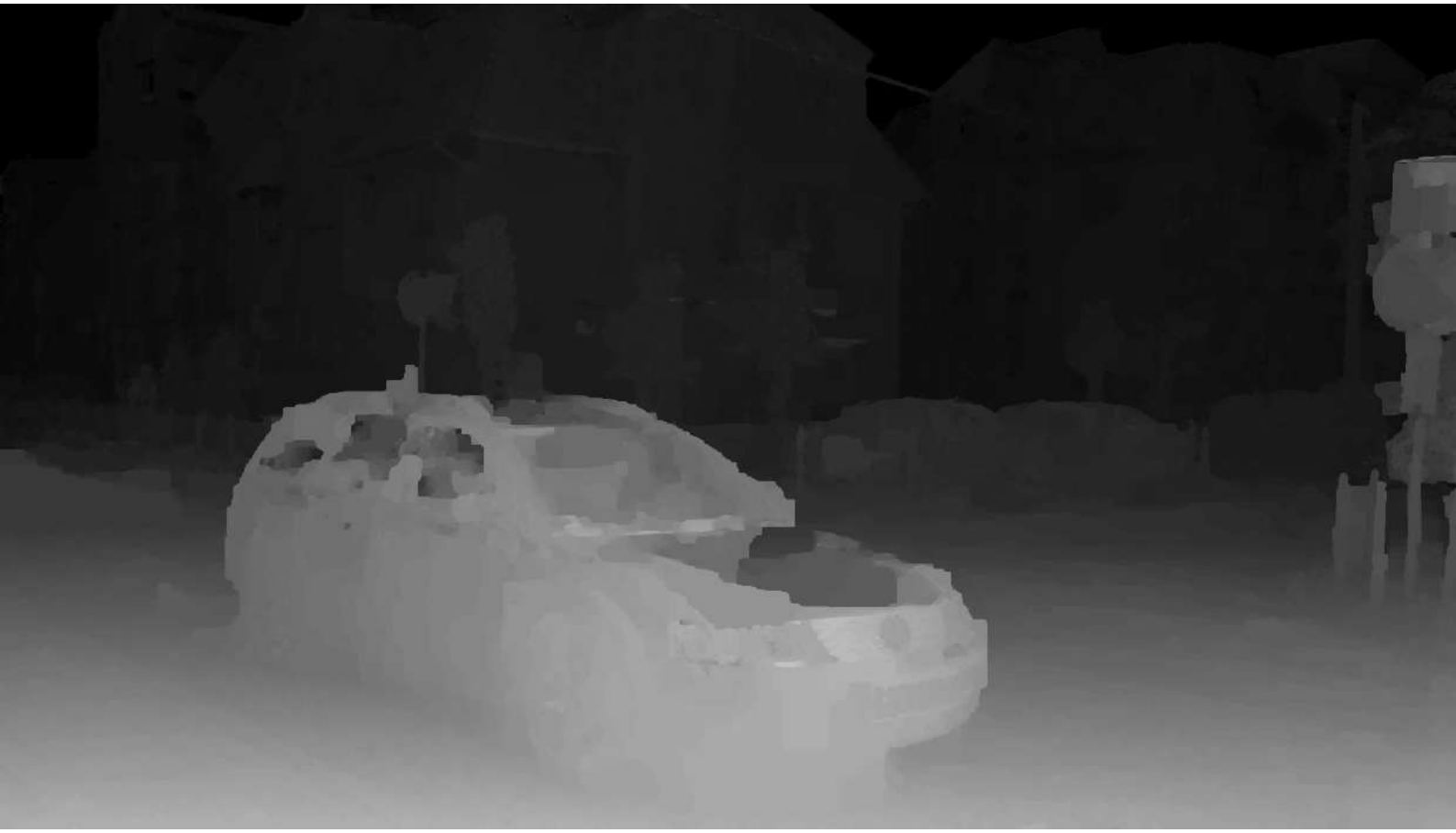

PO 70055



Detailed Report on Depth Image-Based Rendering (DIBR) Algorithm

Mohanna Mousabeigi

August 27, 2024

Introduction

This report provides an in-depth explanation of the Depth Image-Based Rendering (DIBR) algorithm, implemented in Python. The algorithm is designed to generate a virtual view of a scene using an original image, its corresponding depth map, and intrinsic and extrinsic camera parameters. The report details the theoretical background, the implementation process, and the specific steps of the algorithm, supported by the corresponding Python code and explanations.

Algorithm Overview

Depth Image-Based Rendering (DIBR) synthesizes new views of a scene by warping pixels from an existing image based on their depths and the perspective of a virtual camera. The algorithm involves several key stages:

1. **Input Preparation:** Load the original image, depth map, and camera parameters.
2. **Depth Mapping:** Convert depth map values to real-world distances using the Znear and Zfar planes.
3. **3D Point Reconstruction:** Compute the 3D positions of pixels in the original camera's coordinate system.
4. **World Coordinate Transformation:** Transform these 3D points into the world coordinate system.
5. **Virtual Camera Projection:** Project the 3D points onto the image plane of the virtual camera.
6. **Image Warping:** Map the texture from the original image to the newly computed positions.
7. **Artifact Handling and Output Generation:** Handle disocclusions and generate the final image.

Each of these steps is directly linked to specific sections of the Python code provided.

Step 1: Input Preparation

The first step is to load the original image, depth map, and camera parameters. This involves reading the image files and initializing the camera matrices.

```
def load_camera_parameters():
    # Original camera parameters
```

```
    K_original = np.array([[1732.87, 0.0, 943.23],
                           [0.0, 1729.90, 548.845040],
                           [0, 0, 1]])

    Rt_original = np.array([[1.0, 0.0, 0.0, 0],
                            [0.0, 1.0, 0.0, 0.0],
                            [0.0, 0.0, 1.0, 0.0]])

    Znear = 34.506386
    Zfar = 2760.510889

    # Virtual camera parameters
    K_virtual = np.array([[1732.87, 0.0, 943.23],
                          [0.0, 1729.90, 548.845040],
                          [0, 0, 1]])

    Rt_virtual = np.array([[1.0, 0.0, 0.0, 1.5924],
                           [0.0, 1.0, 0.0, 0.0],
                           [0.0, 0.0, 1.0, 0.0]])

    return K_original, Rt_original, Znear, Zfar, K_virtual, Rt_virtual

V_o = cv2.imread('V_original.png')
D_o = cv2.imread('D_original.png', cv2.IMREAD_GRAYSCALE)
K_o, Rt_o, Znear, Zfar, K_v, Rt_v = load_camera_parameters()
```

Listing 1: Loading Input Data and Camera Parameters

Explanation:

- The intrinsic matrix K represents the camera's internal parameters (focal length, principal point).
- The extrinsic matrix Rt represents the camera's orientation and position relative to the world.
- The depth map D_o is loaded as a grayscale image, representing distance from the camera for each pixel.

Step 2: Depth Mapping

Next, the depth values from the depth map are converted to actual Z-coordinates using the Znear and Zfar values.

```
def depth_to_z(depth_map, Znear, Zfar):
    depth_map_normalized = depth_map / 255.0  # Assuming depth map is 8-bit
    grayscale
    Z_o = Znear * Zfar / (Zfar - depth_map_normalized * (Zfar - Znear))
    return Z_o

Z_o = depth_to_z(D_o, Znear, Zfar)
```

Listing 2: Depth Mapping

Explanation:

- The depth map is normalized from 0-255 to 0-1.
- The formula $Z_o = \frac{Z_{\text{near}} \cdot Z_{\text{far}}}{Z_{\text{far}} - d \cdot (Z_{\text{far}} - Z_{\text{near}})}$ converts these normalized values into actual distances from the camera.

Step 3: 3D Point Reconstruction

The next step is to reconstruct the 3D coordinates of each pixel in the original camera's coordinate system.

```
xx, yy = np.meshgrid(np.arange(w), np.arange(h))
X_o = (xx - K_o[0, 2]) * Z_o / K_o[0, 0]
Y_o = (yy - K_o[1, 2]) * Z_o / K_o[1, 1]
P_o = np.vstack((X_o.flatten(), Y_o.flatten(), Z_o.flatten()))
```

Listing 3: 3D Point Reconstruction

Explanation:

- Using the intrinsic matrix K_o, we map each pixel (u, v) to its corresponding 3D point (X, Y, Z).
- The pixel coordinates (u, v) are shifted by the principal point and scaled by the depth value to compute the 3D coordinates.

Step 4: World Coordinate Transformation

The 3D points are then transformed from the camera's coordinate system to the world coordinate system.

```
R_o = Rt_o[:3, :3]
t_o = Rt_o[:3, 3].reshape(3, 1)
M_w = R_o @ P_o + t_o
```

Listing 4: World Coordinate Transformation

Explanation:

- The rotation matrix R_o and translation vector t_o are extracted from the extrinsic matrix.
- The transformation $M_w = R_o \cdot P_o + t_o$ converts the 3D points into the world coordinate system.

Step 5: Virtual Camera Projection

These world coordinates are then projected onto the image plane of the virtual camera.

```
R_v = Rt_v[:3, :3]
t_v = Rt_v[:3, 3].reshape(3, 1)
M_v = R_v @ M_w + t_v

m_v = K_v @ M_v
m_v /= m_v[2, :]  # Normalize by the z-coordinate

x_v = m_v[0, :].reshape(h, w)
y_v = m_v[1, :].reshape(h, w)
```

Listing 5: Virtual Camera Projection

Explanation:

- The 3D points are first transformed by the virtual camera's extrinsic parameters.
- The intrinsic matrix K_v is then applied to project these points onto the virtual image plane.
- The pixel coordinates are normalized to ensure they correspond to valid image positions.

Step 6: Image Warping

Finally, the texture from the original image is mapped onto the newly computed positions in the virtual view.

```
V_v = np.zeros_like(V_o)
valid_mask = np.zeros((h, w), dtype=bool)
for i in range(3):  # Assuming V_o is a color image with 3 channels
    V_v[:, :, i] = map_coordinates(V_o[:, :, i], [y_v, x_v], order=1, mode='
    nearest')
    valid_mask |= (y_v >= 0) & (y_v < h) & (x_v >= 0) & (x_v < w)
```

Listing 6: Image Warping

Explanation:

- The `map_coordinates` function is used to interpolate the pixel values from the original image based on the new positions computed for the virtual view.
- A validity mask is created to track valid pixel locations for further artifact handling.

Step 7: Artifact Handling and Output Generation

The final step involves filling any disocclusions and saving the generated virtual view as an output image.

```
def fill_disocclusions(image, mask):
    kernel = np.ones((5, 5), np.uint8)
    dilated_mask = binary_dilation(mask, structure=kernel).astype(np.uint8)
    filled_image = cv2.inpaint(image, dilated_mask, 3, cv2.INPAINT_TELEA)
    return filled_image

V_v_filled = fill_disocclusions(V_v, ~valid_mask)
output_path = 'V_virtual_filled.png'
cv2.imwrite(output_path, V_v_filled)
```

Listing 7: Handling Artifacts and Saving the Output Image

Explanation:

- The inpainting method fills disocclusions and cracks to enhance visual quality.
- The final image is saved as `V_virtual_filled.png`, representing the scene from the virtual camera's perspective.

Conclusion

This report has detailed each step of the Depth Image-Based Rendering algorithm, linking the theoretical background with the corresponding Python code. The code provided implements the necessary mathematical operations to transform the original image and depth map into a new view from a different camera position. It also incorporates artifact handling techniques to address disocclusions and cracks, ensuring higher visual quality in the synthesized view. The rigorous application of camera transformation principles ensures that the synthesized view is geometrically consistent with the given inputs. The explanations and code comments make the algorithm transparent and easily understandable for further development or application in related research areas.