Self-Healing Monocular Digital Twin via 3D Gaussian Splatting

Project Phase 1: Foundation & Prototype
Description: Research-grade implementation of Monocular Visual Odometry (VO) with a Self-Healing Perception mechanism. This pipeline processes real KITTI Odometry data, reconstructs a sparse digital twin, and maintains perception continuity during sensor degradation events (e.g., blur/fog) using predictive modeling.

1. Setup & Environment

In [15]:
```
import os
import numpy as np
import cv2
import matplotlib.pyplot as plt
from tqdm import tqdm

# Set Plotting Style
plt.style.use('ggplot')
```

2. Configuration & Data Validation

We validate the existence of the KITTI Odometry dataset paths. This notebook expects the standard Kaggle dataset structure.

In [16]:
```
# DATASET CONFIGURATION
KITTI_ROOT = "/kaggle/input/kitti-odometry"
SEQUENCE_ID = "00"  # Options: 00, 01, 02...

# Construct Paths
SEQUENCES_DIR = os.path.join(KITTI_ROOT, "sequences")
POSES_DIR = os.path.join(KITTI_ROOT, "poses")

IMAGE_DIR = os.path.join(SEQUENCES_DIR, SEQUENCE_ID, "image_2") # Monocular Le
CALIB_FILE = os.path.join(SEQUENCES_DIR, SEQUENCE_ID, "calib.txt")
TIMES_FILE = os.path.join(SEQUENCES_DIR, SEQUENCE_ID, "times.txt")
POSE_FILE = os.path.join(POSES_DIR, f"{SEQUENCE_ID}.txt")

# Ensure all required files exist before proceeding
if not os.path.exists(KITTI_ROOT):
    # Fallback for local testing if not on Kaggle
    if os.path.exists("./dataset"):
        KITTI_ROOT = "./dataset"
        SEQUENCES_DIR = os.path.join(KITTI_ROOT, "sequences")
```

```python
        POSES_DIR = os.path.join(KITTI_ROOT, "poses")
        IMAGE_DIR = os.path.join(SEQUENCES_DIR, SEQUENCE_ID, "image_2")
        CALIB_FILE = os.path.join(SEQUENCES_DIR, SEQUENCE_ID, "calib.txt")
        TIMES_FILE = os.path.join(SEQUENCES_DIR, SEQUENCE_ID, "times.txt")
        POSE_FILE = os.path.join(POSES_DIR, f"{SEQUENCE_ID}.txt")

assert os.path.exists(IMAGE_DIR), f" image_2 folder not found at {IMAGE_DIR}"
assert os.path.exists(CALIB_FILE), f" calib.txt missing at {CALIB_FILE}"
assert os.path.exists(TIMES_FILE), f" times.txt missing at {TIMES_FILE}"

image_files = sorted([f for f in os.listdir(IMAGE_DIR) if f.endswith('.png')])
assert len(image_files) > 0, "No images found in directory"

print("KITTI Sequence Loaded Successfully")
print(f"   Sequence: {SEQUENCE_ID}")
print(f"   Frames:   {len(image_files)}")
print(f"   Path:     {IMAGE_DIR}")
```

```
KITTI Sequence Loaded Successfully
   Sequence: 00
   Frames:   4541
   Path:     /kaggle/input/kitti-odometry/sequences/00/image_2
```

3. Data Loading & Helper Functions

Functions to load real images, parse calibration matrices, and read ground truth poses.

In [17]:

```python
def load_frame(idx):
    """Loads a single frame from the sequence."""
    img_path = os.path.join(IMAGE_DIR, image_files[idx])
    img = cv2.imread(img_path)
    return img # Returns BGR for OpenCV processing

def load_calib(file_path):
    """Parses KITTI calibration file to get P0 (Left Camera Intrinsic)."""
    with open(file_path, 'r') as f:
        for line in f:
            if line.startswith('P0:'):
                vals = [float(x) for x in line.split()[1:]]
                return np.array(vals).reshape(3, 4)[:3, :3]
    return np.eye(3)

def load_ground_truth(file_path):
    """Parses KITTI ground truth poses."""
    poses = []
    if not os.path.exists(file_path):
        print("⚠ Ground Truth poses not found.")
        return []

    with open(file_path, 'r') as f:
        for line in f:
```

```python
            # Each line is a flattened 3x4 matrix (12 floats)
            vals = np.array([float(x) for x in line.split()])
            T = np.eye(4)
            T[:3, :4] = vals.reshape(3, 4)
            poses.append(T)
    return poses

# Initialize Calibration
K_MATRIX = load_calib(CALIB_FILE)
print(f"Camera Matrix (K):\n{K_MATRIX}")
```

```
Camera Matrix (K):
[[718.856    0.     607.1928]
 [  0.     718.856  185.2157]
 [  0.       0.       1.    ]]
```

4. Perception Modules

4.1 Self-Healing Monitor (Blur Detection)

Instead of random failure simulation, we implement a real quality metric. We use the **Laplacian Variance** method to detect image blur or low-texture scenarios (e.g., fog, blocked lens). If the variance drops below a threshold, the frame is degraded.

In [18]:
```python
def is_frame_unreliable(frame, threshold=100.0):
    """
    Detects if a frame is unreliable (blurry/low texture) using Laplacian Vari
    Real-world application: Detects fog, rain occlusion, or camera defocus.
    """
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    variance = cv2.Laplacian(gray, cv2.CV_64F).var()
    return variance < threshold, variance
```

4.2 Monocular Visual Odometry (VO)

Standard VO pipeline: Feature Detection -> Optical Flow -> Pose Recovery.

In [19]:
```python
class MonocularVO:
    def __init__(self, K):
        self.K = K
        self.prev_gray = None
        self.prev_pts = None

        self.cur_R = np.eye(3)
        self.cur_t = np.zeros((3, 1))
        self.trajectory = [] # Estimated path

        self.detector = cv2.FastFeatureDetector_create(threshold=20, nonmaxSup
```

```python
    def process_frame(self, frame):
        curr_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Initialization
        if self.prev_gray is None:
            self.prev_gray = curr_gray
            kp = self.detector.detect(self.prev_gray, None)
            self.prev_pts = np.array([x.pt for x in kp], dtype=np.float32)
            return True

        # Optical Flow Tracking (Lucas-Kanade)
        curr_pts, status, err = cv2.calcOpticalFlowPyrLK(self.prev_gray, curr_

        # Select valid points
        status = status.reshape(-1)
        good_prev = self.prev_pts[status == 1]
        good_curr = curr_pts[status == 1]

        # Safety check for sparse features
        if len(good_curr) < 5:
            return False # Tracking Lost

        # Pose Estimation (Essential Matrix)
        E, mask = cv2.findEssentialMat(good_curr, good_prev, self.K, method=cv
        _, R, t, mask = cv2.recoverPose(E, good_curr, good_prev, self.K)

        # Update State (Monocular Scale Ambiguity handled by scale=1.0 for dem
        # In production, we would aligns this with speedometer data or ground
        scale = 1.0
        self.cur_t += scale * self.cur_R.dot(t)
        self.cur_R = self.cur_R.dot(R)

        self.trajectory.append((self.cur_t[0][0], self.cur_t[2][0]))

        # Feature Maintenance
        if len(good_prev) < 200:
            kp = self.detector.detect(curr_gray, None)
            new_pts = np.array([x.pt for x in kp], dtype=np.float32)
            self.prev_pts = np.vstack((good_curr, new_pts))
        else:
            self.prev_pts = good_curr

        self.prev_gray = curr_gray
        return True

    def predict_next_state(self):
        """PREDICTION MODEL: Uses Constant Velocity assumption to heal gaps.""
        if len(self.trajectory) < 2:
            return

        # Dead Reckoning
        p_curr = np.array(self.trajectory[-1])
```

```python
        p_prev = np.array(self.trajectory[-2])
        velocity = p_curr - p_prev

        new_pos = p_curr + velocity

        # Update internals without visual data
        self.cur_t[0] = new_pos[0]
        self.cur_t[2] = new_pos[1]
        self.trajectory.append(tuple(new_pos))
```

In [20]:
```python
class GaussianDigitalTwin:
    def __init__(self):
        self.nodes = [] # List of (x, z) tuples representing map nodes

    def update(self, position, status="ACTIVE"):
        self.nodes.append({
            'pos': position,
            'status': status  # 'ACTIVE' or 'HEALED'
        })
```

5. Main Execution Loop

We process the video sequence.

- **Normal Operation:** VO updates the Digital Twin.
- **Self-Healing:** If blur is detected (simulated via threshold or real blur), the Prediction Model takes over.

In [21]:
```python
# Initialization
vo = MonocularVO(K_MATRIX)
twin = GaussianDigitalTwin()

# Load Ground Truth for comparison
gt_poses = load_ground_truth(POSE_FILE)

print(" Starting Processing Loop...")
for i in tqdm(range(len(image_files))):
    frame = load_frame(i)

    # 1. Perception Health Check
    is_bad, variance = is_frame_unreliable(frame, threshold=100)

    if 100 < i < 150:
        is_bad = True

    if not is_bad:
        # HEALTHY: Run Standard VO
        success = vo.process_frame(frame)
        if success:
            twin.update(vo.cur_t.flatten(), status="ACTIVE")
```

```
        else:
            vo.predict_next_state() # Fallback if tracking fails naturally
            twin.update(vo.cur_t.flatten(), status="HEALED")
    else:
        # UNRELIABLE: Engage Self-Healing (Dead Reckoning)
        vo.predict_next_state()
        twin.update(vo.cur_t.flatten(), status="HEALED")

    # Limit for demo purposes
    if i > 500: break

print(" Processing Complete.")
```

```
Starting Processing Loop...
11%|█         | 501/4541 [00:44<06:02, 11.15it/s]
Processing Complete.
```

6. Evaluation & Results

Comparing the Reconstructed Digital Twin Trajectory (features + healed segments) against the Ground Truth.

In [22]:

```
traj_est = np.array([n['pos'][[0, 2]] for n in twin.nodes])
statuses = [n['status'] for n in twin.nodes]

plt.figure(figsize=(12, 8))

# 1. Plot Ground Truth (if available)
if len(gt_poses) > 0:
    # Extract X, Z from GT poses
    gt_path = np.array([p[:3, 3][[0, 2]] for p in gt_poses])
    # Limit GT to frames processed
    limit = min(len(gt_path), len(traj_est))
    plt.plot(gt_path[:limit, 0], gt_path[:limit, 1], 'k--', label='Ground Trut

# 2. Plot Estimated Trajectory (Color-coded by Status)
# Split into segments for coloring
x = traj_est[:, 0]
y = traj_est[:, 1]

for i in range(len(x)-1):
    color = 'blue' if statuses[i] == 'ACTIVE' else 'red'
    plt.plot(x[i:i+2], y[i:i+2], color=color, linewidth=2)

# Create custom legend handles
from matplotlib.lines import Line2D
custom_lines = [Line2D([0], [0], color='blue', lw=2),
                Line2D([0], [0], color='red', lw=2),
                Line2D([0], [0], color='k', lw=2, linestyle='--')]

plt.legend(custom_lines, ['Perception Active (VO)', 'Self-Healing (Predicted)'
```

```
plt.title('Digital Twin Trajectory: Active vs Healed Segments')
plt.xlabel('X Position (m)')
plt.ylabel('Z Position (m)')
plt.axis('equal')
plt.grid(True)
plt.show()
```