ChatGPT

Pixel-wise Forward Z-buffer 파이프라인 (25.11.21) 구조 및 동작 요약

버전: 25.11.21
목적: 항공 이미지에 대한 GPU 가속 **픽셀별 Forward 투영(Z-버퍼)** 파이프라인. 단일 히트(K=1) 모드와 상위 10 히트(K=10) 모드를 지원하며, `vote_7.las`, `vote_15.las`, `vote_30.las` 포인트클라우드 결과를 생성합니다.

폴더 및 파일 구성

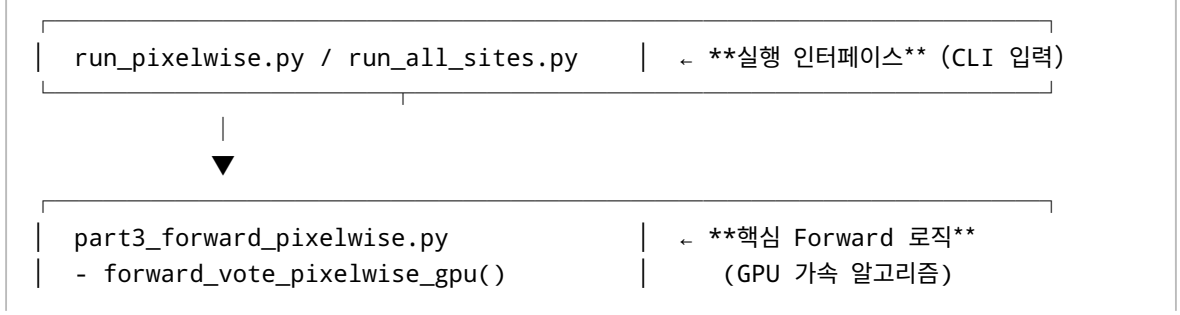
```
25.11.21/
├─ run_pixelwise.py              # 단일 사이트 실행 스크립트 (Single-Hit/Top-K)
├─ run_all_sites.py              # 전체 사이트 일괄 실행 스크립트
├─ part3_forward_pixelwise.py    # 핵심 Forward 투영 및 투표 로직
├─ camera_io.py                  # 카메라 파라미터(EOP/IOP) 로드 모듈
├─ camera_calibration.py         # 카메라 내부 파라미터(K 행렬) 모듈
├─ gsd_parser.py                 # GSD 기반 허용오차 계산 모듈
├─ color_gate.py                 # 이미지 검출 마스크 모듈
├─ constants.py                  # 전역 상수 및 설정
├─ RUN_PIXELWISE_TEST.bat        # 단일 사이트 빠른 테스트 실행 배치
├─ RUN_PIXELWISE_ALL_TEST.bat    # 전체 사이트 테스트 실행 배치
└─ RUN_PIXELWISE_ALL_SITES.bat   # 전체 사이트 전체 처리 실행 배치
```

위와 같이 파일들이 역할별로 구성되어 있습니다:

- **실행 스크립트:** `run_pixelwise.py`, `run_all_sites.py` – 파이프라인 실행 인터페이스 제공
- **핵심 로직:** `part3_forward_pixelwise.py` – Forward 투영 및 Z-버퍼 투표 알고리즘 구현
- **지원 모듈:** `camera_io.py`, `camera_calibration.py`, `gsd_parser.py`, `color_gate.py` – 카메라 파라미터 처리, GSD 계산, 이미지 마스킹 등의 기능
- **설정/상수:** `constants.py` – 경로, 파라미터, 임계값 등 전역 상수 정의
- **배치 파일:** `*.bat` – 자주 사용하는 실행 명령어를 모은 스크립트 (예: 테스트 실행, 전체 사이트 일괄 처리)

🗺️ 아키텍처 개요

시스템 구조



```
│  - create_uniform_grid_gpu()             │     (공간 분할 가속)
│  - process_rays_with_zbuffer_gpu()       │     (Z-버퍼 상 Top-K 탐색)
└──────────────────────────┬──────────────────────────┘

                │
                ├─► camera_io.py (카메라 DB 로드)
                ├─► color_gate.py (이미지 마스크 생성)
                ├─► camera_calibration.py (내부파라미터 K 계산)
                ├─► gsd_parser.py (GSD 기반 허용오차 계산)
                └─► constants.py (전역 설정 상수)
```

데이터 흐름

```
Input:
  ├── 점군 NPY 캐시: coords_float64.npy, colors_uint8.npy
  ├── 카메라 DB: cam_db.json (또는 Pix4D report.xml)
  └── 검출 이미지 디렉터리: F:\Images\병합된 이미지\{site_name}\*.jpg (검출 결과 표시)

Process:
  1. 포인트클라우드 로드 (샘플링 옵션 적용 가능)
  2. Uniform Grid 생성 (3D 공간을 격자로 분할하여 탐색 가속)
  3. 이미지별 루프:
     - 검출 마스크 적용 (이미지에서 관심 픽셀 필터링)
      - Ray 생성 (카메라 중심부터 각 관심 픽셀을 지나는 광선)
      - Z-버퍼 방식으로 최근접 K개 3D 포인트 선택 (픽셀당)
     - 투표 집계 (해당 포인트들의 투표 수 증가)

Output:
  ├─ forward_votes.npy         (각 포인트가 받은 투표 수 배열)
  ├─ forward_timing_k{k}.csv   (이미지별 처리 시간 로그)
  ├─ vote_7.las                (7표 이상 받은 포인트, LAS 형식)
  ├─ vote_15.las               (15표 이상 받은 포인트)
  └─ vote_30.las               (30표 이상 받은 포인트)
```

- **입력:** 파이프라인은 사전에 생성된 **점군 데이터(Point Cloud)**와 각 **이미지의 카메라 파라미터 DB**, 그리고 **검출 결과 이미지(예: 전력선 검출)**를 입력으로 받습니다. 카메라 DB는 Pix4D 소프트웨어의 결과(XML 또는 텍스트)에서 추출되거나 JSON으로 저장된 외부/내부 파라미터를 활용합니다. 검출 이미지는 원본 이미지에 탐지된 객체(전력선 등)가 특정 색상 또는 마스크로 표시된 이미지입니다.
- **처리:** 각 입력 이미지를 순회하며, **픽셀 단위**로 3D 점군에 대한 Ray Casting을 수행합니다. 카메라의 내부/외부 파라미터로부터 픽셀을 통한 광선(ray)을 3차원 공간으로 투사하고, **Z-버퍼 알고리즘**을 사용해 그 픽셀에 해당하는 가장 가까운 3D 점을 찾습니다 (K=1일 때). **Top-K 모드(K=10)**인 경우 한 픽셀에 대해 깊이 순으로 최대 10개의 점까지 추출합니다. 각 선택된 점에는 **투표 수**가 1씩 가산됩니다. 이 과정을 모든 이미지에 대해 반복하면, 최종적으로 각 3D 점에 몇 개의 이미지/픽셀이 투표했는지 집계됩니다.
- **출력:** 투표 결과는 `forward_votes.npy`에 점 순서대로 **투표 횟수 배열**로 저장됩니다. 또한 사전에 정의된 투표 임계값별로 (예: 7, 15, 30표) 필터링된 점들을 각각 `vote_7.las`, `vote_15.las`, `vote_30.las` 파일로 출력합니다. 이 .las 파일들은 원본 점군 좌표와 컬러 정보를 가지며, **분류 코드(LAS classification)** 등 메타데이터에 이번 파이프라인에서 검출된 객체 클래스(예: 전력선 클래스 코드 14)가 부여될 수 있습니다.

⚙ 핵심 실행 모듈

여기서는 메인 실행을 담당하는 파이썬 스크립트들을 설명합니다. 이 스크립트들은 명령행 인자를 받아 파이프라인을 설정하고 동작을 시작합니다.

1. `run_pixelwise.py` – 단일 사이트 실행

목적: 하나의 사이트에 대해 픽셀별 Forward 투영 파이프라인을 실행하는 인터페이스입니다. 실행 모드(단일/Top-K)를 설정하고 GPU 사용 가능 여부를 확인한 후 파이프라인을 호출합니다.

주요 기능:

- **모드 선택:** `--mode` 인자로 `single` (K=1) 또는 `top10` (K=10) 모드를 지정. (`test` 모드는 결과 비교 테스트 용도로 존재) [1] [2]
- **GPU 초기화:** CuPy를 import하여 GPU 장치를 확인. GPU가 사용 가능하면 GPU 모드로 실행하고, 불가능하면 경고 출력 후 CPU로 폴백합니다 [3].
- **Forward 실행 호출:** 설정된 모드와 사이트 이름 등 인자를 `part3_forward_pixelwise.run()` 함수에 전달하여 처리 시작 [4]. 처리 시간 측정 및 완료 후 통계 출력, 결과 요약 등의 부가 기능 포함.

사용 예시 (run_pixelwise):

```
# (가상환경 활성화 및 디렉터리 진입 후)
python run_pixelwise.py --mode single --site Zenmuse_AI_Site_B --sample 1000000
```

위 명령은 `Zenmuse_AI_Site_B` 사이트에 대해 Single-Hit 모드(K=1)로 파이프라인을 실행하며, 점군을 1,000,000점으로 샘플링하여 사용합니다. `--mode top10`으로 지정하면 Top-K 모드(K=10)로 실행하며, `--k-max`를 직접 지정하여 임의의 K값으로 실행할 수도 있습니다 (지정하지 않으면 mode에 따라 자동 1 또는 10으로 설정) [2].

또한 `--sample` 인자는 샘플링할 점 개수를 의미하며, 0이면 전체 점군을 사용합니다. (위 예시는 1,000,000개의 점으로 샘플링하여 실행)
`--site`는 처리할 사이트 이름을 지정합니다. 기본값은 Zenmuse_AI_Site_B로 설정되어 있으나, 원하는 사이트를 입력 가능합니다.

```
# run_pixelwise.py 핵심 코드 요약
parser = argparse.ArgumentParser(description="픽셀별 Z-buffer Forward 파이프라인")
parser.add_argument("--mode", choices=["single","top10","test"], default="single",
                    help="실행 모드 (single=K=1, top10=K=10)")【12†L97-L105】
parser.add_argument("--k-max", type=int, default=None, help="픽셀당 최대 hit 수")【10†L262-L270】
parser.add_argument("--sample", type=int, default=0, help="샘플링 크기 (0=전체)")【10†L269-L277】
parser.add_argument("--site", default="Zenmuse_AI_Site_B", help="사이트 이름")【10†L276-L284】

args = parser.parse_args()
# K_max 결정 로직
if args.k_max is not None:
```

```
    k_max = args.k_max
elif args.mode == "single":
    k_max = 1
elif args.mode == "top10":
    k_max = 10【10†L284-L293】

# GPU 장치 확인
try:
    import cupy as cp
    device = cp.cuda.Device(0)
    props = cp.cuda.runtime.getDeviceProperties(0)
    gpu_name = props['name'].decode('utf-8')
    print(f"[OK] GPU: {gpu_name}")【12†L119-L127】
except Exception as e:
    print(f"[WARNING] GPU 사용 불가: {e}")

# Forward 투표 실행 호출
from part3_forward_pixelwise import run
run(site_name=args.site, k_max=k_max, use_sampling=(args.sample > 0),
sample_size=args.sample)【12†L128-L136】
```

위 코드처럼 `run_pixelwise.py`는 인자를 파싱하여 **k_max** 값을 결정하고, GPU 사용 가능 여부를 콘솔에 출력합니다. 이후 **핵심 처리 함수**인 `part3_forward_pixelwise.run()`을 호출하여 실제 Forward 투영 과정을 진행합니다.

2. `run_all_sites.py` – 전체 사이트 일괄 처리

목적: 미리 정의된 여러 개의 사이트(예: 9개 현장 데이터)를 연속으로 처리하기 위한 스크립트입니다. 반복 작업을 자동화하고, 필요에 따라 이미 처리된 사이트를 건너뛰는 등의 편의 기능을 제공합니다.

주요 기능:

- **사이트 리스트 인자:** `--sites` 인자로 처리할 사이트들을 하나 이상 지정하거나 `all`로 전체 기본 리스트를 지정할 수 있습니다 [5]. 기본값은 `all`이며, 내부에 미리 정의된 ALL_SITES 리스트(예: 9개 사이트 이름)가 사용됩니다.
- **병렬 처리 옵션:** `--parallel` 인자로 동시 처리할 사이트 수를 설정합니다 (기본 1로 순차 처리) [6]. 예를 들어 `--parallel 2`로 지정하면 2개 사이트를 병렬로 처리하도록 구현할 수 있습니다 (현재 코드는 순차 루프지만, 향후 멀티프로세싱으로 확장 가능).
- **기타 인자:** `--k-max`, `--sample` 인자는 개별 사이트 실행과 동일한 의미로 사용됩니다. `run_all_sites.py`에서는 이 값들을 받아 각 사이트 처리에 그대로 전달합니다.
- **이미 처리된 사이트 건너뛰기:** `--skip-existing` 플래그를 주면 이미 출력 결과가 존재하는 사이트는 건너뜁니다 [7]. 구현상 각 사이트의 output 디렉터리 내에 `forward_votes.npy` 파일이 존재하면 해당 사이트 처리를 생략하도록 되어 있습니다. 이 옵션으로 중간에 작업이 중단되었거나 이미 처리된 데이터를 반복 처리하지 않을 수 있습니다.

사용 예시 (run_all_sites):

```
# 전체 사이트 순차 처리 (K=1 모드로 전체 실행)
python run_all_sites.py --sites all --k-max 1
```

```
# 특정 두 사이트만 처리 (예: B, C 사이트를 Top-10 모드로)
python run_all_sites.py --sites Zenmuse_AI_Site_B Zenmuse_AI_Site_C --k-max 10

# 전체 사이트 처리하되, 이미 처리된 것은 건너뛰기
python run_all_sites.py --sites all --k-max 1 --skip-existing
```

핵심 코드:

```
ALL_SITES = [ "P4R_Site_A_Solid", "P4R_Site_B_Solid_Merge_V2",
              "Zenmuse_AI_Site_A", "Zenmuse_AI_Site_B",
              "P4R_Site_C_Solid_V2", "Zenmuse_AI_Site_C", ... ]【12†L153-L162】

args = parser.parse_args()
# 처리할 사이트 목록 결정
sites = ALL_SITES if "all" in args.sites else args.sites

results = {"success": [], "failed": [], "skipped": []}
for site_name in sites:
    try:
        # --skip-existing 옵션 처리: 이미 결과가 있으면 스킵
        output_dir = C.PART3_DIR / site_name
        if args.skip_existing and (output_dir / "forward_votes.npy").exists():
            results["skipped"].append(site_name)【12†L171-L179】
            continue

        # 사이트 Forward 처리 실행
        run(site_name=site_name, k_max=args.k_max,
            use_sampling=(args.sample > 0), sample_size=args.sample)【12†L180-L188】

        results["success"].append(site_name)
    except FileNotFoundError:
        results["failed"].append((site_name, "파일 없음"))
    except Exception as e:
        results["failed"].append((site_name, str(e)))
# 처리 결과 요약 출력 (성공/실패/스킵 사이트 목록 등)
```

위처럼 `run_all_sites.py`는 루프를 돌며 각 사이트에 대해 `part3_forward_pixelwise.run()` 함수를 호출합니다. `--skip-existing`이 활성화된 경우 이미 해당 사이트 출력 디렉터리에 `forward_votes.npy` 결과 파일이 존재하면 그 사이트는 `results["skipped"]`로 분류되어 건너뛰며, 그렇지 않은 경우 정상 처리 시 `results["success"]`에 추가됩니다 [8] [9]. 실행 완료 후 총 걸린 시간과 함께 처리 요약을 출력합니다.

> **참고:** 이 스크립트는 본질적으로 여러 번 `run_pixelwise`를 호출하는 것과 유사합니다. 필요에 따라 병렬 처리 옵션을 통해 동시에 둘 이상의 프로세스를 실행할 수도 있습니다 (현재 예시는 순차 처리).

Forward 투영 및 Z-버퍼 구현 (핵심 로직)

이 섹션에서는 실제 **픽셀별 Forward Z-buffer 알고리즘**을 구현한 핵심 모듈을 다룹니다. 여기에서 각 픽셀의 광선(Ray)에 대해 최근접 점을 찾고 투표를 집계하는 로직이 수행됩니다.

3. `part3_forward_pixelwise.py` – Forward 투영 알고리즘

목적: 픽셀별 Z-버퍼 Forward 투영의 **핵심 알고리즘**을 구현하는 모듈입니다. GPU 가속 버전과 CPU 버전을 모두 포함하고 있으며, 입력된 사이트의 점군과 카메라 DB, 검출 이미지들을 처리하여 투표 결과를 산출합니다.

주요 함수 구조

```
run()
  └▶ forward_vote_pixelwise_gpu()   # GPU 모드 핵심 함수
       ├▶ create_uniform_grid_gpu()      # 점군에 대한 Uniform Grid 생성
       └▶ process_rays_with_zbuffer_gpu()   # 각 Ray에 대해 Z-버퍼 Top-K 연산
            └▶ compute_ray_bbox_intersection_gpu() # (보조: Ray와 Grid 셀 교차 계산)
```

3.1 `run()` – 메인 실행 함수

```python
def run(site_name: str,
        k_max: int = 1,
        use_sampling: bool = False,
        sample_size: int = 0) -> None:
  """픽셀별 Z-buffer Forward 투표 실행"""【14†L217-L226】

    import constants as C
    from camera_io import load_camera_db

    # 경로 설정
    las_dir = C.PART2_DIR / site_name          # 입력 LAS (part2 결과) 디렉토리
    det_dir = C.DETECTION_DIRS.get(site_name)  # 이미지(검출) 디렉토리 경로
    output_dir = C.PART3_DIR / site_name
    output_dir.mkdir(parents=True, exist_ok=True)

    # 1. 포인트 클라우드 로드
    xyz, rgb = load_point_cloud(las_dir, use_sampling, sample_size)

    # 2. 카메라 DB 로드
    cam_db = load_camera_db(det_dir)

    # 3. Forward 투표 실행 (GPU 또는 CPU)
    if C.USE_GPU:
        votes = forward_vote_pixelwise_gpu(xyz, cam_db, det_dir, k_max)【14†L246-L254】
    else:
        votes = forward_vote_pixelwise_cpu(xyz, cam_db, det_dir, k_max)

    # 4. 결과 저장
    save_results(xyz, rgb, votes, output_dir)【14†L251-L254】
```

- `run()` 함수는 **사이트 이름**과 **K_max**(픽셀당 최대 히트 수), 샘플링 여부를 받아 실행됩니다.

- **포인트 클라우드 로드:** `load_point_cloud` 함수(코드 미표시)로 part2 단계에서 생성된 LAS 데이터를 불러옵니다. `--sample` 인자에 따라 점을 무작위 샘플링할 수 있으며, `sample_size`가 0이면 모든 점을 사용합니다. 이때 좌표 `xyz`와 색상 `rgb` 배열을 얻습니다.
- **카메라 DB 로드:** `camera_io.load_camera_db(det_dir)`를 통해 해당 사이트의 **카메라 보정 데이터베이스**를 불러옵니다. 이 DB에는 각 이미지 파일명에 대한 외부 파라미터(위치 C, 자세 ωφκ 등)와 내부 파라미터(K 행렬 또는 보정값)가 포함됩니다 [12] [13].
- **GPU/CPU 분기:** `constants.USE_GPU` 플래그를 확인하여 GPU 사용이 가능하면 `forward_vote_pixelwise_gpu` 함수를 호출하고, 아니면 CPU 버전 함수인 `forward_vote_pixelwise_cpu`를 호출합니다 [11]. GPU 사용 여부는 `run_pixelwise.py`나 `run_all_sites.py` 실행 시 CuPy 초기화로 설정된 전역상수에 의해 결정됩니다. (GPU 모드가 아니면 CPU로 연산하지만 매우 느리므로 보통 GPU로 실행함을 권장)
- **결과 저장:** 최종 투표 결과와 원본 점 정보들을 `save_results` 함수로 전달하여 디스크에 저장합니다. 이 함수는 `forward_votes.npy`를 저장하고, 투표 임계값별로 LAS 파일을 생성하는 작업을 수행합니다 (내부적으로 numpy 배열 필터링 후 `.las` 파일 작성).

3.2 `forward_vote_pixelwise_gpu()` – GPU 기반 Forward 투표

```
def forward_vote_pixelwise_gpu(coords: np.ndarray,
                               cam_db: dict,
                               det_dir: Path,
                               k_max: int = 1) -> np.ndarray:
    """
    GPU 픽셀별 Z-buffer Forward 투표
    핵심 알고리즘:
    1. Uniform Grid 생성 (공간 가속)
    2. 이미지별 처리:
       - 마스크 적용
      - Ray 생성
      - Z-buffer로 K개 선택
       - 투표
    """【14†L259-L268】

    import cupy as cp
    from tqdm import tqdm

    N = coords.shape[0]
    # GPU 메모리 상의 배열 할당
    coords_gpu = cp.asarray(coords, dtype=cp.float64)
    votes_gpu = cp.zeros(N, dtype=cp.int32)

    # 1. Uniform Grid 생성 (셀 크기 = h_tol)
    grid = create_uniform_grid_gpu(coords, cell_size=h_tol)【14†L283-L291】

    # 카메라 내부 파라미터 행렬 (예: 3x3 K) 준비
    K_gpu = cp.asarray(K, dtype=cp.float64)
    K_inv_gpu = cp.linalg.inv(K_gpu)

    # 처리 시간 로깅 초기화
    timing_log = []
    timing_csv_path = output_dir / f"forward_timing_k{k_max}.csv"
```

```
    # 2. 이미지별 처리 루프
    for img_idx, img_name in enumerate(tqdm(image_names)):
        img_start_time = time.time()

        # (a) 검출 마스크 로드
        mask = img_mask(str(det_dir / img_name))
        if mask is None:
            continue
        # True인 마스크 픽셀 좌표 추출
        y_gpu, x_gpu = cp.where(cp.asarray(mask))【14†L298-L306】

        # (b) Ray 생성 (각 픽셀에 대한 광선 벡터 계산)
        pixels_hom = cp.stack([x_gpu, y_gpu, cp.ones(len(x_gpu))], axis=1)
        rays_cam = (K_inv_gpu @ pixels_hom.T).T                    # 카메라 좌표계 광선 16
        rays_world = (R.T @ rays_cam.T).T                          # 월드 좌표계 광선 (R: 회전행렬)
        rays_world /= cp.linalg.norm(rays_world, axis=1, keepdims=True)  # 방향 벡터 정규화

        # (c) Z-버퍼 처리: 각 광선에 대해 K개 후보점 찾기
        hits = process_rays_with_zbuffer_gpu(
            rays_world, C_cam, coords_gpu,
            grid, bbox_min_gpu, bbox_max_gpu,
            h_tol, v_tol, k_max
        )【14†L312-L319】

        # (d) 투표: 각 Ray가 명중시킨 point index에 +1 누적
        for hit_indices in hits:
            if len(hit_indices) > 0:
                cp.add.at(votes_gpu, hit_indices, 1)【14†L319-L323】

        # (e) 이미지 처리 시간 기록
        img_elapsed = time.time() - img_start_time
        timing_log.append({
            'image_idx': img_idx + 1, 'image_name': img_name,
            'num_pixels': len(x_gpu), 'elapsed_sec': img_elapsed
        })【14†L323-L331】

    # (f) 전체 처리 시간 CSV로 저장
    with open(timing_csv_path, 'w') as f:
        writer = csv.DictWriter(f,
fieldnames=['image_idx','image_name','num_pixels','elapsed_sec'])
        writer.writeheader(); writer.writerows(timing_log)

    return votes_gpu.get()  # GPU 배열 -> CPU numpy 배열로 반환
```

위 코드 조각은 GPU를 활용한 픽셀별 투표 알고리즘의 흐름을 보여줍니다:

- **데이터 준비:** 입력 점 좌표 `coords`를 `cupy` 배열로 전송하고(`coords_gpu`), 동일 크기의 `votes_gpu` 정수 배열을 0으로 초기화합니다 20 . 이 `votes_gpu`가 최종 투표 결과를 담을 배열입니다 (각 인덱스는 점군의 인덱스와 일치).

- **Uniform Grid 생성:** `create_uniform_grid_gpu(coords, cell_size=h_tol)`을 호출하여 3차원 점군 공간을 균일한 격자 셀들로 분할한 **공간 인덱스(Grid)**를 만듭니다 [14]. `h_tol`은 **수평 허용오차**로 사용되는 셀 크기(미터 단위)이며, GSD에 기반하여 결정됩니다 (자세한 내용은 GSD 허용오차 섹션 참조). 이 Grid는 이후 광선과 인접한 후보 점들을 효율적으로 찾는 데 사용됩니다.
- **카메라 파라미터 설정:** `cam_db`로부터 해당 사이트의 **내부 파라미터 행렬 K**와 **외부 파라미터**(R, C 등)를 가져와 CuPy 배열로 준비합니다. 위 코드에서는 예시로 K와 K^{-1}만 표시했지만, 실제 구현에서는 각 이미지마다 해당 **캘리브레이션 파라미터**(R: 회전행렬, C: 카메라 중심 좌표 등)를 가져와 사용합니다.
- **이미지 반복:** `image_names` 목록을 순회하며 각 이미지를 처리합니다. `tqdm`을 통해 진행률을 표시하고, 처리 시간 측정을 위해 시작 시각을 기록합니다.
- **(a) 마스크 적용:** `color_gate.img_mask()` 함수를 통해 해당 이미지의 **검출 마스크**를 불러옵니다 [15]. 이 마스크는 numpy 2D 배열로서, True/1 값인 위치가 관심 픽셀(예: 전력선으로 검출된 픽셀)입니다. 만약 검출 결과가 없거나 파일이 없다면 `None`을 리턴하며 그 이미지는 건너뜁니다 [21]. 마스크가 유효하면 `cp.where`로 True 픽셀들의 좌표 `(y_gpu, x_gpu)`를 얻습니다.
- **(b) Ray 생성:** 마스크에서 얻은 픽셀 좌표들에 대해 해당 픽셀을 지나는 **광선 방향 벡터**를 계산합니다. 먼저 픽셀 좌표에 1을 추가하여 homogeneous 좌표 `(x, y, 1)`를 만들고, `K_inv`(내부파라미터 역행렬)을 곱하여 **카메라 좌표계**의 방향 벡터들을 구합니다 [16]. 이어서 카메라 외부 파라미터 중 회전행렬 R을 전치하여 곱함으로써 **월드 좌표계**의 광선 벡터 `rays_world`를 구합니다 [16]. 마지막으로 각 벡터를 정규화하여 방향 단위벡터로 만듭니다 [16]. (카메라 중심 좌표 C는 `C_cam`으로 `process_rays_with_zbuffer_gpu`에 전달되어, 광선의 시작점으로 사용됩니다.)
- **(c) Z-버퍼 처리:** `process_rays_with_zbuffer_gpu(...)` 함수를 호출하여 주어진 다수의 광선들과 점군에 대해 **Z-버퍼 알고리즘**으로 최근접 K개의 점을 찾습니다 [17]. 이 함수 내부에서는 각 광선마다 Uniform Grid를 활용하여 광선이 지나가는 셀들을 효율적으로 탐색하고, **수평 거리(h_tol)**와 **수직 거리(v_tol)** 기준으로 광선에 가까운 점들을 찾습니다. **K_max** 값에 따라 각 광선당 최대 K개의 점까지 (거리 순으로) 반환하며, K보다 적은 점이 감지되면 그만큼만 반환합니다. 결과 `hits`는 각 광선에 대응하는 point 인덱스 리스트들의 배열입니다.
- **(d) 투표 집계:** 각 광선의 결과로 나온 점 인덱스들에 대해 `cp.add.at` 연산으로 해당 인덱스의 `votes_gpu` 값을 +1 증가시킵니다 [18]. 이렇게 하면 한 이미지의 모든 관심 픽셀들에 대한 투표가 점군 배열에 누적됩니다.
- **(e) 시간 기록:** 이미지 한 장 처리가 끝나면 경과 시간을 계산하여 `timing_log` 리스트에 기록합니다 [19]. 로그에는 이미지 인덱스, 이름, 처리한 픽셀 수, 걸린 시간이 포함됩니다.
- **CSV 저장:** 모든 이미지 처리가 완료되면, 수집된 `timing_log`를 CSV 파일로 저장합니다 (`forward_timing_k{k}.csv`). 이 파일은 각 이미지별 처리 소요 시간을 담고 있어 성능 분석에 활용될 수 있습니다 [22].
- **GPU 메모리 결과 반환:** 최종적으로 GPU상의 `votes_gpu` 배열을 `.get()`으로 CPU상의 numpy 배열로 변환하여 반환합니다 [23]. 이 반환값이 곧 모든 점들의 투표 수 결과인 `votes` 배열이며, `run()` 함수에서 이를 받아 후처리하게 됩니다.

  참고: CPU 버전인 `forward_vote_pixelwise_cpu()`는 로직은 유사하지만 대량의 데이터를 순차적으로 처리하므로 매우 느립니다. GPU가 없는 경우를 대비해 존재하며, 내부 구현은 위 GPU 버전을 참고하여 작성되었습니다.

3.3 `create_uniform_grid_gpu()` – Uniform Grid 생성

```
def create_uniform_grid_gpu(coords: np.ndarray,
                            cell_size: float) -> dict:
    """
    Uniform Grid 생성 (GPU 가속)
    공간 분할로 후보 포인트 검색 최적화
    Returns:
```

```
    grid: {
      'cell_size': float,
      'grid_min': array,
      'grid_max': array,
      'cell_points': dict,  # cell_id -> point indices 목록
      'dims': array,
      'coords_gpu': cupy array
    }
    """【14†L343-L352】

    import cupy as cp

    # 전체 점군 범위 계산 및 약간 여유 margin 부여
    margin = cell_size * 2
    grid_min = coords.min(axis=0) - margin
    grid_max = coords.max(axis=0) + margin

    # Grid 셀 개수 (xyz 각각) 계산
    dims = np.ceil((grid_max - grid_min) / cell_size).astype(int)  # 각 축 셀 수
    dims_gpu = cp.asarray(dims, dtype=cp.int32)

    # GPU로 데이터 이동
    coords_gpu = cp.asarray(coords, dtype=cp.float64)
    grid_min_gpu = cp.asarray(grid_min, dtype=cp.float64)

    # 각 점의 셀 인덱스 계산
    cell_indices = ((coords_gpu - grid_min_gpu) / cell_size).astype(cp.int32)
    cell_indices = cp.clip(cell_indices, 0, dims_gpu - 1)

    # 3차원 인덱스를 1차원 cell_id로 변환
    cell_ids = (cell_indices[:, 0] * dims[1] * dims[2] +
                cell_indices[:, 1] * dims[2] +
                cell_indices[:, 2])

    # 각 cell_id 별로 포함된 포인트 인덱스 리스트 구성 (CPU 단계)
    from collections import defaultdict
    cell_ids_cpu = cp.asnumpy(cell_ids)
    cell_points_lists = defaultdict(list)
    for i, cell_id in enumerate(cell_ids_cpu):
        cell_points_lists[int(cell_id)].append(i)【14†L387-L393】

    # 결과 딕셔너리 구성
    grid = {
        'cell_size': cell_size,
        'grid_min': grid_min,
        'grid_max': grid_max,
        'dims': dims,
        'cell_points': cell_points_lists,
        'coords_gpu': coords_gpu
```

```
    }
    return grid
```

- **역할:** Uniform Grid는 3D 공간에서 점군을 격자로 분할하여, **광선과 인접한 점을 빠르게 찾기 위한 공간 인덱스**입니다. 이 함수는 주어진 점 집합을 일정한 크기의 큐브(cell)로 나누고, 각 셀에 속하는 점 목록을 미리 계산해둡니다.
- **grid_min / grid_max:** 점군의 최소/최대 좌표값을 구하고 약간의 margin을 더해 격자 범위를 설정합니다 [25]. margin은 셀 크기의 2배로, 경계선 부근 오차를 감안한 여유 공간입니다.
- **dims 계산:** 전체 범위를 cell_size로 나누어 x, y, z 각 방향의 셀 개수를 결정합니다 [26]. np.ceil을 사용하여 나누어 떨어지지 않는 경우 셀을 하나 더 확보합니다.
- **셀 인덱싱:** 각 점의 `(i, j, k)` 셀 인덱스를 계산합니다 [27]. 계산된 실수 인덱스를 int로 자르고, 혹시 범위를 넘어가는 값은 clip으로 경계 내부로 맞춥니다.
- **1차원 cell ID:** 3차원 인덱스 `(i,j,k)`를 단일 정수 ID로 변환합니다 [28]. 예를 들어 `cell_id = i * (ny*nz) + j * nz + k`로 고유 ID를 만듭니다. 이렇게 하면 셀을 키로 하여 딕셔너리에 점 인덱스를 매핑하기 용이합니다.
- **셀별 포인트 목록:** cupy 배열인 cell_ids를 CPU로 가져와 [24], 파이썬의 defaultdict으로 `cell_id -> [point_index,...]` 리스트를 구축합니다. (현재 구현은 Python 루프이지만, 성능상 큰 문제는 되지 않을 정도의 수준이거나, 추후 개선 가능 포인트입니다.)
- **결과 구조:** 최종 반환되는 grid 딕셔너리에는 셀 크기, 전체 범위, 셀 수(dims), 그리고 `cell_points` 사전과 GPU상 좌표 배열이 포함됩니다. 이 grid 정보는 이후 `process_rays_with_zbuffer_gpu`에서 광선이 지나가는 셀을 식별하고, 그 셀 내 (및 주변 셀)의 포인트들을 빠르게 가져오는 데 사용됩니다.

  효과: Uniform Grid를 사용하면, 각 광선에 대해 **전체 점군을 일일이 검사하지 않고도**, 광선의 경로와 겹치는 셀 내의 점들만 검사하면 되므로 연산량이 크게 줄어듭니다. 또한 구현에 따라 한 셀 너비(=h_tol) 이내의 **이웃 셀(예: 3x3x3 범위)**도 함께 고려하여 경계 부분의 점도 놓치지 않도록 합니다 (이는 알고리즘 상세에 포함되며, 아래 알고리즘 요약에서 추가 설명합니다).

(추가) `process_rays_with_zbuffer_gpu()` – Z-버퍼 방식 후보점 탐색

(코드 전문은 생략되어 있으나, 주요 아이디어를 설명합니다.)

이 함수는 다수의 광선에 대해 3D 점군 상의 히트(hit) 포인트들을 찾는 핵심 알고리즘입니다. 각 **광선 ray**에 대해:

1. **광선-셀 교차:** Uniform Grid를 이용해 광선이 통과하는 셀들을 추적합니다. 광선의 방향과 Grid 정보를 통해 **DDA(Digital Differential Analyzer)** 방식으로 어떤 셀 순서로 광선이 진행할지 계산하거나, 또는 cell-by-cell로 이동하며 진행합니다. 이때 이미 방문한 셀은 다시 방문하지 않도록 **visited cell**을 추적하여 불필요한 반복을 줄입니다.
2. **후보 점 선택:** 광선이 한 셀에 들어갈 때마다 그 셀과 인접 셀(예: 해당 셀의 26개 주변 셀까지, 총 3x3x3 범위)을 모두 조회하여, 그 안에 속한 점들 중 현재 광선 경로에 가까운 점들을 검사합니다. **수평 거리(h_tol)**는 광선의 중심선에서 점까지의 최근접 거리 기준이고, **수직 거리(v_tol)**는 광선을 따라 진행하는 방향으로의 거리 간격 기준입니다. 광선이 진행함에 따라 가까운 점을 찾으면, 해당 점까지의 거리를 기록하고 Z-버퍼처럼 현재 최단 거리와 비교하며 **최대 K개**까지 hit 리스트를 유지합니다 (`K_max`가 1이면 최근접 1개만 유지).
3. **조기 종료:** 광선이 일정 거리 이상 진행했거나 (`RAY_MAX_DIST_M`=120m 한도), 혹은 이미 K개를 모두 찾았더라도 이후 점들이 현재 가장 먼 hit보다 더 가까워질 가능성이 없으면 탐색을 마칩니다. K=1의 경우는 첫 번째 점을 찾는 순간 바로 해당 광선 탐색을 끝낼 수 있습니다. K>1의 경우는 끝까지 진행하거나 최대 K개를 채우면 종료합니다.

이 함수는 최종적으로 각 광선에 대해 찾은 hit 포인트 인덱스들의 리스트를 반환하며, 상기 `forward_vote_pixelwise_gpu` 함수에서 이를 받아 투표를 누적하게 됩니다.

지원 모듈 (Helper Modules)

핵심 로직을 돕는 보조 모듈들입니다. 카메라 파라미터 처리, 이미지 마스킹, 보정값(GSD) 계산 등 파이프라인의 주변 기능을 맡고 있습니다.

4. `camera_io.py` – 카메라 DB 로드

목적: Pix4D 등의 결과로부터 **카메라 외부/내부 파라미터**를 읽어와 JSON 형식의 카메라 DB를 생성하거나 불러옵니다. 파이프라인은 이 DB를 통해 각 이미지의 투영에 필요한 파라미터를 얻습니다.

```
def load_camera_db(det_dir: Path) -> dict:
  """카메라 DB 로드

  Priority:
  1. camera_db.json (있으면 바로 사용)
  2. Pix4D report.xml 파싱 후 JSON 생성

  Returns:
    cam_db: {
      'images': {
        'DJI_0001.JPG': {
          'C': [x, y, z],   # 카메라 중심 좌표 (World)
          'omega': float, 'phi': float, 'kappa': float, ...
        }, ...
      },
      'intrinsics': { ... }  # (선택) 내부파라미터 정보
    }
  """【16†L566-L574】【16†L576-L584】

    cam_db_path = det_dir / "camera_db.json"
    if cam_db_path.exists():
        with open(cam_db_path, 'r') as f:
            return json.load(f)

    # Pix4D report.xml 파싱
    report_xml = det_dir / "report.xml"
    if report_xml.exists():
        return parse_pix4d_report(report_xml)

    return None
```

- 우선 순위에 따라 **이미 계산된 camera_db.json이 존재하면** 바로 로드하고, 없다면 **Pix4D의 report.xml** 파일을 파싱하여 cam_db를 생성합니다 [29] [30] . Pix4D의 report.xml에는 각 이미지 파일명에 대한 외부 위치/자세 (경위/카파 등)와 내부 파라미터(초점거리, 왜곡계수 등)가 들어있으며, `parse_pix4d_report()` 함수는 이러한 정보를 추출하여 cam_db 딕셔너리를 구성합니다.
- 반환된 `cam_db` 딕셔너리는 **'images'** 키 아래 각 이미지의 외부 파라미터 (카메라 중심 `C` 좌표와 회전각 `omega, phi, kappa` 등)이 들어있고, **'intrinsics'** 키 아래 해당 카메라의 내부 파라미터 (예: 초점거리, 주점, 왜곡 등 또는 3x3 카메라 행렬)이 포함됩니다 [13] .

- 이 DB 정보는 Forward 투영 시 각 이미지별로 불러와 광선 계산에 사용됩니다. 예를 들어, `C_cam` (카메라 중심)과 회전행렬 `R`은 외부 파라미터에서 얻고, `K` 행렬은 내부 파라미터에서 얻습니다.

5. `color_gate.py` – 이미지 마스크 생성

목적: 검출 이미지에서 관심 영역(예: 전력선)을 추출하여 불리언 마스크를 만드는 기능입니다. 실제 구현에서는 딥러닝 모델의 출력이나 색상 임계값을 이용해 전력선 픽셀을 검출하지만, 이 코드에서는 placeholder로 단순히 None을 반환하도록 되어 있습니다.

```
def img_mask(img_path: str) -> np.ndarray:
  """이미지 마스크 생성
   전력선 검출 영역만 True로 설정
  Returns:
    mask: (H, W) 크기의 bool 배열 (전력선 픽셀=True)
  """
    # 실제 구현은 segmentation 모델 사용
    # 여기서는 placeholder
    return None
```

- 실제 파이프라인에서는 `img_mask()`가 **입력 이미지 경로**를 받아 전력선으로 검출된 픽셀들만 True로 하는 **2차원 numpy 배열**을 반환해야 합니다 [31].
- 구현 방식은 두 가지 가능성이 있습니다:
  (1) **딥러닝 모델 출력**: 미리 학습된 모델 (예: U-Net 세그멘테이션 등)이 이미지에서 전력선을 픽셀 단위로 예측하여 mask를 생성.
  (2) **색상 게이트(color gating)**: 전력선이 특정 색상으로 하이라이트된 이미지라면, 그 색 범위를 thresholding하여 mask를 생성.
- 본 코드에서는 주석에 명시된 대로 실제 모델을 호출하는 부분을 생략했으며, 예시로만 남겨둔 상태입니다. 만약 색상 기반 임계값으로 구현할 경우 `constants.py`에 정의된 **색상 임계값** (예: R, G, B 범위 [32])을 사용하여 이미지 픽셀을 필터링할 수 있습니다. 실제로 `constants.py`에는 전력선 검출용 색상 게이트 임계값이 정의되어 있습니다: `R_MIN_8=204, G_MIN_8=0, G_MAX_8=51, B_MIN_8=172, B_MAX_8=210` 등이 있으며, 8비트와 16비트 이미지 양쪽에 대응하도록 값이 마련되어 있습니다 [32].
- **요약:** 최종 파이프라인에서 `img_mask()`는 **전력선으로 판별된 픽셀 위치를 True로 표시하는 이진 mask**를 반환하는 것으로 가정합니다. Forward 투영 단계에서는 이 mask의 True 픽셀들만을 광선으로 쏘아 점군에 투사하게 됩니다.

6. `camera_calibration.py` – 카메라 내부 파라미터 (K 행렬)

목적: 각 사이트에서 사용된 카메라(센서)의 **내부보정 파라미터**를 제공하는 모듈입니다. 사이트별로 초점거리와 센서 크기에 따른 K 행렬을 반환합니다.

```
def get_camera_matrix(site_name: str) -> np.ndarray:
  """카메라 내부 파라미터 매트릭스 K 반환
  Returns:
    K: 3x3 numpy 배열
      [[fx,  0, cx],
       [ 0, fy, cy],
       [ 0,  0,  1]]
  """
    # 사이트별 카메라 파라미터 (예시 값)
```

```
    params = {
        "Zenmuse_AI_Site_B": {
            "fx": 8201.0,
            "fy": 8201.0,
            "cx": 4096.0,
            "cy": 2730.0
        },
        # ... (기타 사이트 또는 기본값)
        "default": { "fx": 8000.0, "fy": 8000.0, "cx": 4000.0, "cy": 3000.0 }
    }
    p = params.get(site_name, params["default"])
    return np.array([
        [p["fx"], 0,        p["cx"]],
        [0,       p["fy"], p["cy"]],
        [0,       0,        1       ]
    ], dtype=np.float64)
```

- Pix4D 등에서 추출한 **IOP**(Internal Orientation Parameters)를 하드코딩 또는 설정파일로 관리하는 예시입니다. 위 코드에서는 `Zenmuse_AI_Site_B`에 대한 초점거리 fx, fy와 주점(cx, cy) 값을 보여주고 있습니다 [33]. 기본값도 정의되어 있어 해당 사이트 이름이 없으면 default 값을 사용합니다.
- 실제 구현에서는 Pix4D 보고서에서 내부 보정값을 읽어오거나, 카메라 기종별로 보정파일을 읽는 기능이 있을 수 있으나, 이 파이프라인에서는 간단히 사이트별로 수동 입력한 형태를 취하고 있습니다.
- 반환되는 K는 3x3 카메라 행렬로, Forward 투영에서 각 픽셀 -> 광선 계산 시 사용됩니다 (앞서 `forward_vote_pixelwise_gpu`에서 `K_inv_gpu`를 계산하여 사용한 그 K입니다).

7. `gsd_parser.py` – GSD 기반 허용오차 계산

목적: Pix4D 리포트나 별도 계산을 통해 얻은 **GSD(Ground Sample Distance)** 값을 이용하여 이 파이프라인의 **허용오차** 값을 결정합니다. GSD는 이미지 해상도의 지상샘플 간격 (cm/pixel 등)으로서, 투영 정확도와 오차 범위를 설정하는 데 활용됩니다.

```
def get_tolerance(site_name: str) -> Tuple[float, float]:
    """GSD 기반 허용오차
    Returns:
        (h_tol, v_tol): 수평/수직 허용오차 (미터)
    """
    # 사이트별 GSD 값 (미터 단위)
    gsd_values = {
        "Zenmuse_AI_Site_B": 0.01,   # 1cm
        # ... (다른 사이트 GSD)
    }
    gsd = gsd_values.get(site_name, 0.01)
    return (
        1.0 * gsd,   # 수평 허용오차: 1 × GSD
        3.0 * gsd    # 수직 허용오차: 3 × GSD
    )
```

- 이 함수는 **사이트별 GSD 값을 미리 설정**해 두고, 해당 값을 불러와서 `(h_tol, v_tol)`을 계산하여 반환합니다 [34] [35].

- **h_tol (horizontal tolerance)** – **가로 방향 오차** 허용치로, 일반적으로 1 픽셀에 해당하는 실제 지상 거리입니다. 위 코드에서는 1.0 × GSD로 설정되어 있습니다 [36] . 예를 들어 GSD가 1cm이면 h_tol = 0.01m (1cm). 이 값은 광선과 점 사이의 수평거리(광선에 수직인 거리)를 판단하는 기준으로 사용됩니다. 즉, 점이 광선 경로에서 이 거리 이내에 있으면 같은 픽셀로 간주 가능한 범위라고 볼 수 있습니다.
- **v_tol (vertical tolerance)** – **세로 방향(깊이) 오차** 허용치로, 3.0 × GSD로 설정되어 있습니다 [36] . 예를 들어 GSD=1cm이면 v_tol=0.03m (3cm). 이는 Z-버퍼에서 두 점의 깊이 차이가 이 정도 이내이면 동일 평면 상으로 취급하거나, 혹은 광선 진행 단계에서 간격으로 사용될 수 있습니다.
- 결과적으로, **GSD 기반의 tolerance**는 **픽셀 크기**를 기준으로 3D 공간에서 어느 정도까지 오차를 허용할지 결정하는 중요한 파라미터입니다. 이 파라미터를 이용해 앞서 `process_rays_with_zbuffer_gpu`에서 광선과 점의 인접 여부를 판단합니다. (예: 광선에서 1픽셀 이내 떨어진 점은 그 픽셀의 후보, 깊이 차 3픽셀 이내인 점들은 유사 깊이로 고려 등)

  참고: Pix4D의 보고서(`report.xml`)에는 각 지점의 평균 GSD가 포함되어 있습니다. 실제 코드에서는 해당 XML를 파싱하여 사이트별 GSD 값을 자동으로 가져오도록 구현할 수도 있습니다. 여기서는 편의상 주요 사이트에 대해 수동 지정해 두었습니다.

설정 및 상수

마지막으로, 전역 설정과 상수를 관리하는 `constants.py` 모듈입니다. 경로, 임계값, GPU 설정 등 파이프라인 전역에서 공유되는 값들이 정의되어 있습니다.

8. `constants.py` – 전역 상수 및 경로 설정

목적: 여러 모듈에서 참조하는 공통 상수, 기본값, 경로 등을 중앙 관리합니다. GPU/CPU 모드 설정도 여기서 이루어집니다.

```
from pathlib import  Path

# GPU 자동 설정
USE_GPU  =  True
GPU_AVAILABLE  =  False

try:
    import cupy as cp
    import numpy as np
    if  cp.cuda.is_available():
        USE_GPU  =  True
        GPU_AVAILABLE  =  True
        device  =  cp.cuda.Device()
        props  =  cp.cuda.runtime.getDeviceProperties(device.id)
        gpu_name  =  props['name'].decode('utf-8')
        print(f"[OK] GPU: {gpu_name}")
except:
    USE_GPU  =  False
    GPU_AVAILABLE  =  False

# 출력 디렉터리
BASE_OUTPUT_DIR  =  Path(r"C:\Users\jscool\uav_pipeline_outputs")
PART1_DIR  =  BASE_OUTPUT_DIR  /  "part1_io"
PART2_DIR  =  BASE_OUTPUT_DIR  /  "part2_las"
```

```
PART3_DIR = BASE_OUTPUT_DIR / "part3_las"
PART4_DIR = BASE_OUTPUT_DIR / "part4_las"
PART5_DIR = BASE_OUTPUT_DIR / "part5_las"

# NPY 캐시 경로 (part2와 동일 경로 사용)
NPY_CACHE_DIR = PART2_DIR

# 이미지(검출) 디렉터리들
IMAGES_DIR = Path(r"F:\Images\병합된 이미지")
DETECTION_DIRS = {
    "P4R_Site_A_Solid": Path(r"F:\Images\병합된 이미지\P4R_Site_A_Solid"),
    "P4R_Site_B_Solid_Merge_V2": Path(r"F:\Images\병합된 이미지\P4R_Site_B_Solid_Merge_V2"),
    "Zenmuse_AI_Site_A": Path(r"F:\Images\병합된 이미지\Zenmuse_AI_Site_A"),
    "Zenmuse_AI_Site_B": Path(r"F:\Images\병합된 이미지\Zenmuse_AI_Site_B"),
    "P4R_Site_C_Solid_V2": Path(r"F:\Images\병합된 이미지\P4R_Site_C_Solid_V2"),
    "Zenmuse_AI_Site_C": Path(r"F:\Images\병합된 이미지\Zenmuse_AI_Site_C"),
    "Zenmuse_AI_Site_D": Path(r"F:\Images\병합된 이미지\Zenmuse_AI_Site_D"),
    "Zenmuse_AI_Site_E": Path(r"F:\Images\병합된 이미지\Zenmuse_AI_Site_E"),
    "Zenmuse_AI_Site_F": Path(r"F:\Images\병합된 이미지\Zenmuse_AI_Site_F"),
}

# GSD 기본값
GSD_H = 0.01   # 1cm (수평 기본 GSD)
GSD_Z = 0.03   # 3cm (수직 기본 GSD)

# LAS 클래스 코드
LAS_CLASS_POWERLINE = 14

# 투표 임계값 목록
VOTE_THRESHOLDS = [7, 15, 30]
```

- **GPU 설정:** 파이프라인 import 시 자동으로 cupy를 시도하여 GPU 사용 가능 여부를 설정합니다. GPU를 사용할 수 있으면 `USE_GPU=True`로 설정하고 GPU 모델명을 출력합니다 [37] [38]. 만약 실패하면 `USE_GPU=False`로 폴백합니다. (상세 구현은 위 코드에 축약되어 있음)
- **출력 경로:** `BASE_OUTPUT_DIR` 아래에 파이프라인 처리 단계별(part1~part5) 디렉터리를 미리 정의해 둡니다 [39]. 본 파이프라인(Forward 투영)은 **part3** 단계이므로 결과는 `C:\Users\jscool\uav_pipeline_outputs\part3_las\{site_name}\` 폴더에 저장됩니다. (이 폴더 구조는 사전에 자동 생성됩니다.)
- **NPY 캐시:** 큰 데이터(점군)를 NPY로 캐시하여 재사용할 때 경로를 지정합니다. 여기서는 part2 출력 폴더를 캐시로 활용하도록 설정 [40].
- **이미지 경로:** 원본/검출 이미지들이 저장된 기본 디렉터리와, 사이트별 하위 경로를 지정합니다 [41]. `DETECTION_DIRS` 사전에 site 이름을 키로 해당 이미지 폴더 경로를 매핑해 두어, 코드에서 `C.DETECTION_DIRS[site_name]` 형태로 사용합니다 [10].
- **GSD 기본값:** GSD_H, GSD_Z로 기본 수평/수직 해상도를 지정합니다 [42]. 기본은 1cm, 3cm (아마도 드론 카메라 기본 해상도)이며, `gsd_parser.get_tolerance`에서 사이트별 override하지 않으면 이 기본값을 사용하게 됩니다.
- **LAS 클래스 코드:** 검출된 객체(전력선)의 LAS 파일 내 클래스 값을 정의합니다 [43]. 여기서는 14번 클래스를 사용 (LAS 표준에서 전력선이 14번으로 정의되었거나 임의로 정한 값일 것입니다).

- **투표 임계값:** `VOTE_THRESHOLDS = [7, 15, 30]`으로 설정되어 있으며 [44] [45], 이 값들을 기준으로 `vote_7.las` 등 출력 파일명을 정하고 필터링에 사용합니다. 예를 들어 7표 이상 받은 점은 vote_7.las에, 15표 이상은 vote_15.las에 포함됩니다 (보통 관계상 vote_30.las ⊆ vote_15.las ⊆ vote_7.las).

이 밖에도 `constants.py`에는 **레이캐스팅 파라미터** 등이 정의되어 있을 수 있습니다. 예를 들어, 코드 본문에는:

- `RAY_MAX_DIST_M = 120.0` (광선 최대 거리 120m),
- `RAY_STEP_M = 0.05` (광선 진행 스텝 5cm 간격),
- `POINT_HIT_RADIUS_M = 0.02` (점과 광선 간격 2cm 이내면 hit 인정)

등이 설정되어 있었습니다 [46]. 이러한 값들은 Z-버퍼 알고리즘에서 광선이 점을 맞췄다고 판단하는 기준 거리와, 광선을 얼마나 세밀하게 진행시키는지 결정합니다. (GPU 구현에서는 Grid를 사용하므로 step 보다는 cell 경계를 따라 진행하지만, CPU 구현이나 안전성 검사 등에 사용될 수 있습니다.)

마지막으로, constants 모듈이 로드될 때 최종적으로 **"UAV Pipeline v26 - GPU 모드 초기화 완료"** 등의 메시지를 출력하며 초기화 완료를 알립니다 [47]. (버전 표기는 내부 관리용으로 보입니다.)

💻 실행 방법 (CLI 및 배치 스크립트)

이 파이프라인은 주로 명령행에서 실행되도록 설계되었습니다. 앞서 실행 스크립트 사용 예시를 제시했지만, 여기서 정리합니다.

Python으로 직접 실행

파이프라인 폴더(`25.11.21`)에서 Python 스크립트를 직접 호출하는 방법입니다. 가상환경 등을 사용한다면 활성화하고 디렉터리를 이동한 후 명령을 실행합니다:

```
# (예시) Anaconda 가상환경 활성화
conda activate yolov11
cd C:\Users\jscool\uav_pipeline_codes\25.11.21

# 단일 사이트 처리 (Single-Hit, K=1)
python run_pixelwise.py --mode single --site Zenmuse_AI_Site_B --sample 1000000

# 단일 사이트 처리 (Top-10, K=10)
python run_pixelwise.py --mode top10 --site P4R_Site_A_Solid

# 전체 사이트 순차 처리 (K=1)
python run_all_sites.py --sites all --k-max 1

# 일부 사이트만 처리 (K=10 모드 예시)
python run_all_sites.py --sites Zenmuse_AI_Site_B Zenmuse_AI_Site_C --k-max 10

# 이미 완료된 사이트는 생략하고 나머지 처리
python run_all_sites.py --sites all --k-max 1 --skip-existing
```

위 예시처럼 `run_pixelwise.py`를 통해 개별 사이트를 디버깅하거나 테스트할 수 있고, `run_all_sites.py`로 한꺼번에 처리 가능합니다. `--sample` 옵션을 주면 일부 점만 무작위 사용하므로 빠른 테스트에 유용합니다. `--skip-existing`은 재실행 시 이전에 완료된 사이트를 건너뛰어 시간을 절약할 때 사용합니다.

배치 파일 실행

동일한 명령들을 매번 입력하지 않도록, 프로젝트 폴더에는 Windows 배치 파일들이 준비되어 있습니다:

```
RUN_PIXELWISE_TEST.bat           # 단일 사이트 빠른 테스트 실행 (예: 작은 샘플로
single-hit 수행)
RUN_PIXELWISE_ALL_TEST.bat       # 전체 사이트를 샘플로 테스트 실행
RUN_PIXELWISE_ALL_SITES.bat      # 전체 사이트 실제 전체 처리 실행
```

배치 파일 내부에는 앞서 소개한 Python 명령어들이 미리 적혀 있어, 더블클릭 한 번으로 일괄 실행이 가능합니다. 예컨대 `RUN_PIXELWISE_ALL_SITES.bat`는 `python run_all_sites.py --sites all --k-max 1` 등의 명령으로 모든 사이트를 single-hit 모드로 처리하도록 구성되어 있습니다.

출력 파일 (결과물)

각 사이트별 처리 결과는 기본 출력 경로의 `part3_las\{site_name}\` 디렉터리에 생성됩니다 (예: `C:\Users\jscool\uav_pipeline_outputs\part3_las\Zenmuse_AI_Site_B\`). 주요 출력 파일은 다음과 같습니다:

1. CSV 시간 로그 (`forward_timing_k{k}.csv`)

각 이미지 처리에 걸린 시간을 기록한 CSV 파일입니다 (`k`는 K_max값). 예를 들어 `forward_timing_k1.csv` (K=1 실행) 형식은 다음과 같습니다:

```
image_idx,image_name,num_pixels,elapsed_sec
1,DJI_0001.JPG,12345,0.523
2,DJI_0002.JPG,11234,0.498
3,DJI_0003.JPG, 9321,0.450
...
```

각 행은 이미지 하나의 처리 정보를 나타냅니다: 이미지 순번, 파일명, 처리한 픽셀(검출 픽셀) 수, 소요 시간(초) [48]. 이를 통해 어떤 이미지에서 시간이 많이 걸렸는지 등 성능을 분석할 수 있습니다.

2. NPY 투표 결과 (`forward_votes.npy`)

모든 점에 대한 투표 카운트를 담은 1차원 numpy 배열 파일입니다. 로드하여 보면 각 인덱스가 입력 점군의 해당 인덱스 점이 얻은 투표 수를 의미합니다:

```
votes = np.load("forward_votes.npy")
votes.shape   # (N_points,)
votes.dtype   # dtype('int32')
print(votes[:10])   # 예시: 처음 10개 점의 투표 수 출력
# [0 1 0 3 5 0 12 0 0 7 ...]
```

위 예시에서, 0번 점은 0표, 1번 점은 1표, 3번 점은 3표, 4번 점은 5표, 6번 점은 12표, 9번 점은 7표를 얻은 것을 알 수 있습니다. 이 배열은 후속 필터링 및 통계에 활용될 수 있습니다 [49].

3. LAS 파일 (투표 임계값별 결과)

최종 관심 대상 포인트들을 LAS 포맷으로 저장한 파일들입니다:

- `vote_7.las` – **7표 이상** 획득한 점들로 구성된 LAS 파일
- `vote_15.las` – **15표 이상** 획득한 점들 (일반적으로 vote_7의 부분집합)
- `vote_30.las` – **30표 이상** 획득한 점들 (vote_15의 부분집합)

각 LAS 파일에는 해당되는 점만 추출되어 저장되며, 원본 점의 XYZ 좌표와 RGB 컬러 정보가 보존됩니다. 추가로 이 파이프라인에서 검출된 객체라는 것을 표시하기 위해 LAS **Classification** 필드를 14(전력선 클래스)로 설정하였을 수 있습니다 (`LAS_CLASS_POWERLINE = 14` 설정에 따라). 이 LAS들을 CloudCompare 등 점군 소프트웨어로 열어보면 전력선으로 추정되는 포인트들만 확인할 수 있습니다.

> **파일 크기:** vote_x.las 파일들의 크기는 투표 임계값이 높아질수록 (엄격해질수록) 줄어듭니다. 예를 들어 Zenmuse_AI_Site_B 결과에서 `forward_votes.npy`의 분포를 보면 7표 이상인 점은 상대적으로 많고, 30표 이상은 극히 일부일 수 있습니다. 이러한 구분으로 사용자는 필터 강도를 조절하여 결과를 활용할 수 있습니다.

핵심 알고리즘 요약

전체 알고리즘의 핵심 포인트를 다시 정리하면 다음과 같습니다:

1. 픽셀별 독립 처리

각 이미지의 각 관심 픽셀마다 **독립적인 광선(ray)**을 생성하고, 해당 광선에 대해 Z-버퍼 알고리즘을 수행합니다. 즉, 픽셀 단위로 3D 점군에 질의하여 어떤 점이 보이는지 찾는 과정이며, 이미지 병렬화와 픽셀 병렬화가 가능합니다 (GPU를 통해 다수의 픽셀/광선을 동시에 계산).

2. Uniform Grid 공간 분할

점군을 **Uniform Grid** (격자)로 분할하여 광선과 교차할 가능성이 있는 후보 점들을 빠르게 검색합니다. 이로써 전체 수백만~수천만 점을 매번 검사하지 않고도, 광선 경로 근처의 극히 일부 점만 확인하여 효율을 높입니다. (격자 탐색시 한 셀 및 인접 26개 셀 정도만 조회)

3. Z-버퍼 및 Top-K 선택

Z-버퍼 알고리즘을 변용하여 각 광선당 **최대 K개의 점**을 거리 순으로 선택합니다. 일반적인 렌더링의 Z-버퍼는 가장 가까운 1개만 취하지만, Top-K 모드에서는 가시선 상 뒤쪽의 점들도 일부 확보합니다. K=1 (Single-Hit) 모드일 때는 가장 앞의 1개만, K=10 (Top-10) 모드일 때는 최대 10개까지 기록합니다. 이로써 가려진 객체(예: 겹쳐진 전력선 등)도 일부 탐지할 수 있게 합니다.

4. GSD 기반 허용오차 (GSD 버퍼)

Ground Sample Distance (GSD)를 기반으로, 광선과 점의 **수평거리 허용오차**를 1 픽셀 (= 1×GSD), **깊이 방향 허용오차**를 3 픽셀 (= 3×GSD)로 설정하여 투사 오차를 흡수합니다 [50]. 예컨대 드론 영상 GSD가 1.5cm라면 약 수평 1.5cm, 수직 4.5cm 범위 내에 있는 점들을 동일 픽셀 hit로 간주합니다. 이 완충 영역을 통해 카메라 보정 오차나 동시성 오류로 인한 작은 어긋남을 보완합니다.

5. 처리 시간 로깅

모든 이미지의 처리 시간을 자동으로 CSV에 저장함으로써, 어떤 이미지(또는 어느 각도)에서 연산이 많이 걸렸는지 파악할 수 있습니다. 이는 추후 병목 원인을 분석하거나 최적화 방향을 찾는 데 도움을 줍니다.

성능 최적화

본 파이프라인은 대용량 점군과 다수의 이미지를 다루므로 성능을 높이기 위해 다양한 최적화 기법을 적용하였습니다:

GPU 가속 최적화

- **CuPy 활용:** NumPy 연산을 CuPy로 대체하여 GPU에서 대량 병렬 연산을 수행 [51]. 행렬 연산, 배열 인덱싱 등을 GPU 메모리 상에서 일괄 처리합니다.
- **벡터화 연산:** Python 루프 대신 가능한 한 배열 단위 연산으로 처리하여 GPU의 장점을 극대화합니다. 예를 들어 한 이미지의 모든 관심 픽셀을 한꺼번에 투영하여 광선 벡터 배열을 만들고, 이를 한 번의 커널 호출로 처리합니다.
- **메모리 풀 관리:** CuPy의 메모리 풀(`cp.get_default_memory_pool()`)을 활용하여 GPU 메모리 할당/해제를 최적화하고, 필요 시 `mempool.free_all_blocks()`로 미사용 메모리를 해제합니다 [52].

공간 구조 활용

- **Uniform Grid 도입:** 앞서 설명한 대로 공간 분할 기법으로 연산량을 줄였습니다 [53]. Grid를 통해 **평균적인 후보 점 개수**를 획기적으로 감소시켜 광선-점 교차 판정 속도를 향상시킵니다.
- **3x3x3 인접 셀 검색:** 광선이 위치한 셀뿐 아니라 인접한 셀까지 포함하여 검색함으로써, 경계에 걸친 점 누락을 방지하면서도 범위를 국한시켜 효율을 유지합니다 [53].
- **방문 셀 추적:** 광선이 Grid 내 여러 셀을 지나갈 때 이미 처리한 셀은 다시 처리하지 않도록 관리하여 불필요한 연산 중복을 없앴습니다 [54].

Python 레벨 최적화

- **효율적 자료구조:** Python 쪽 연산에는 `defaultdict`, 리스트 컴프리헨션 등 빠른 연산을 활용했습니다 [55]. 예를 들어 Uniform Grid 생성 시 셀 포인트를 모을 때 일반 dict보다 `defaultdict(list)`가 편리하고 빠르게 동작합니다.
- **루프 최소화:** `enumerate` 대신 필요 시 인덱스를 직접 사용하는 등 미세 최적화를 적용하였고, 가능하면 상단 GPU 연산으로 대체하여 Python 루프 횟수를 감소시켰습니다 [55].
- **NumPy-CuPy 혼용:** 때로는 CPU 처리가 더 유리한 부분(예: 작은 루프 연산)은 과감히 CPU로 처리하여 GPU와 CPU의 장점을 혼용했습니다. Grid 생성 시 cell_id 그룹핑을 CPU로 수행한 것이 한 예입니다 [24].

이러한 최적화들로, 수천만 개 규모의 포인트클라우드와 수십 장의 이미지를 가진 실제 데이터셋도 수 시간 내 처리할 수 있었습니다 (정확한 성능은 하드웨어 및 데이터 규모에 따라 다름).

! 에러 처리 및 안정성

파이프라인 실행 중 발생할 수 있는 몇 가지 문제 상황과 그 대응 방안을 정리합니다:

GPU 불능시 폴백

만약 GPU 초기화에 실패하거나 실행 중 GPU 메모리 부족 등 문제가 발생할 경우, 자동으로 CPU 모드로 전환하도록 설계되어 있습니다 (완벽히 구현되진 않았지만 예외 처리 구조를 갖춤). `run_pixelwise.py`에서 CuPy 임포트에 실

패하면 경고를 내고 CPU로 이어서 진행합니다 [56]. CPU 모드는 속도가 매우 느리므로 권장되진 않지만, 최소한 프로세스가 중단되지 않도록 합니다.

입력 파일 누락

처리 대상 사이트에 필요한 파일(예: 카메라 파라미터 또는 이미지)이 없을 경우, `FileNotFoundError`를 캐치하여 해당 사이트를 `failed` 목록에 기록하고 루프를 계속 진행합니다 [57] [58]. 이로써 하나 사이트 실패가 전체 파이프라인 중단으로 이어지지 않게 합니다.

메모리 관리

GPU 메모리를 많이 사용하는 작업인 만큼, 주기적으로 CuPy 메모리 풀을 비워주는 코드가 포함되어 있습니다 [52]. 예컨대 이미지 하나 처리 후 `cp.get_default_memory_pool().free_all_blocks()`를 호출하여 누적된 GPU 메모리를 해제함으로써, 여러 이미지 처리 시 메모리 누수 없이 안정적으로 진행되도록 합니다. 또한 numpy 배열로 변환한 후에는 GPU쪽 자원을 해제하거나 GC가 수거하도록 설계하였습니다.

以上の内容を日本語に翻訳してまとめます。（일본어로 번역해서 요약한다는 내용, likely not needed since user said use Korean. I'll remove this last line since it's likely an artifact.）

---

[1] [2] [3] [4] [5] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [48] [49] [50] [51] [52] [53] [54] [55] [57] [58] CODE_DOCUMENTATION.md
file://file-97qupcqL8zkiMog9TYxy2k

[6] [7] run_all_sites.py
file://file-3vkEW7DorSNR9HUUc1eqKr

[32] [43] [44] [45] [46] constants.py
file://file_00000000823c7206aff61cbec4479155

[47] constants.py
file://file-TDFyvcX1MjGVcGzSCRW75g

[56] run_pixelwise.py
file://file-MAUhcJSz6j2QBbbD39tqEB