ChatGPT

UAV Z-버퍼 Forward 투표 시스템 개선 기술 명세

문제 원인 및 분석

현재 UAV 기반의 Z-버퍼 Forward 투표 시스템에서 **GPU 연산이 모두 SKIP(건너뜀)**되는 문제가 확인되었습니다. 조사 결과, **모든 입력 영상의 지상 투영 영역(footprint)**이 잘못 계산되어 포인트 클라우드 전체 영역을 덮고 있었고, 이로 인해 알고리즘의 **footprint 기반 필터링** 단계에서 **모든 연산이 생략**되고 있었습니다. 구체적인 원인은 **카메라 고도 계산 오류**입니다. Footprint의 경계(bounding box)를 계산할 때 **고도값을 잘못 설정**하여, 원래는 **카메라 고도 = 카메라의 절대 Z좌표 - 사이트의 평균 지면 높이(ground_Z)**로 써야 하나, 기존 코드에서 `altitude = camera_Z(절대 좌표)`로 직접 사용하고 있었습니다. 예를 들어, Pix4D `report.xml`에서 추출한 해당 사이트의 평균 지면 높이 `ground_Z ≈ 109m`인데, 한 이미지의 EOP상 카메라 **절대 좌표 Z = 189m**를 그대로 고도로 사용하면서 실제 비행 고도는 약 **80m**임에도 불구하고 189m로 간주되었습니다. 이로 인해 **계산된 footprint 영역이 실제보다 훨씬 크게 추정**되어 **포인트 클라우드 전체를 포함**하게 되었고, 시스템은 "이미지가 모든 포인트를 커버한다"고 판단하여 **GPU 처리를 매번 생략**하는 흐름이 발생했습니다. 요약하면, **잘못된 고도 계산으로 인한 footprint 오차**가 **GPU 연산 SKIP 문제**의 근본 원인이었습니다.

Footprint 계산 공식 및 구현

앞서 확인된 문제를 해결하기 위해 `compute_camera_footprint()` **함수**를 리팩토링합니다. 이 함수는 **카메라의 이미지가 지면에 투영되는 영역(footprint)**을 정확히 계산하여 반환합니다. 주요 개선 사항은 **카메라 고도의 정확한 반영**과 **4 코너 기반의 폴리곤 계산**입니다.

- **입력**: 카메라 **내부 파라미터**(초점거리 fx, fy, 주점(cx, cy)), **이미지 크기**(픽셀 단위 너비, 높이), **카메라 자세**(월드 좌표계에서의 회전 행렬 R_c2w), **카메라 중심 좌표 Cw** (월드 좌표계에서의 위치), 그리고 **평균 지면 높이 ground_Z** (사이트별 report.xml에서 얻은 값).

- **카메라 시야각(FOV) 계산**: 내부 파라미터로부터 카메라의 시야각을 계산할 수 있습니다. 예를 들어 **수평 FOV**는 다음과 같습니다:

$$\mathrm{fov}_x = 2 \cdot \arctan\Big(\frac{\text{이미지너비}}{2 \cdot f_x}\Big),$$

  수직 시야각 fov_y도 이와 유사하게 $2\arctan(\text{이미지높이}/(2f_y))$로 계산됩니다 [1]. 이 시야각 계산은 참고용이며, 이후의 정확한 투영 계산은 **영상의 네 구석 픽셀을 이용**하여 수행합니다.

- **4-코너 레이 투영**: 이미지의 **네 모서리 픽셀** (예: (0,0), (W,0), (0,H), (W,H); 좌상, 우상, 좌하, 우하 좌표)을 **카메라 좌표계 기준 시선 방향**으로 변환합니다. 각 코너 픽셀에서 광축을 지나는 **레이(ray)**를 계산하는 절차는 다음과 같습니다:

- **이미지 좌표 -> 카메라 좌표**: 픽셀 좌표를 카메라 좌표계의 방향 벡터로 변환합니다. 예를 들어, 좌상단 픽셀 (u, v) = (0, 0)에 대해, 카메라 좌표계에서의 정규화된 방향 벡터 **d_c** = $[(u - c_x)/f_x,\ (v - c_y)/f_y,\ 1]$^T 로 설정합니다. 이 벡터는 카메라 좌표계에서 해당 픽셀을 향하는 광선 방향을 나타냅니다. 네 모서리 픽셀 각각에 대해 이 계산을 수행합니다.
- **카메라 -> 월드 변환**: 각 **d_c**를 **카메라 좌표계에서 월드 좌표계로 변환**합니다. 회전행렬 R_c2w를 이용하여 **d_w = R_{c2w} \cdot d_c**로 계산하면, 월드 좌표계에서 카메라로부터 뻗어나가는 광선 **방향 벡터 d_w**를 얻습니다. (R_c2w는 **카메라 좌표계의 벡터를 월드 좌표계로 회전**시켜 주는 행렬입니다.)

- **지면 평면과의 교차**: **ground_Z** (지면의 고도, 즉 월드 좌표계에서 기준 지면의 Z값) 평면과 각 광선을 교차시켜 **footprint**의 꼭짓점 좌표를 구합니다. 카메라 중심 Cw = (C_x, C_y, C_z)에서 시작한 광선 $\mathbf{r}(t) = \mathbf{C}_w + t \cdot \mathbf{d}_w$가 **Z = ground_Z** 평면과 만나는 매개변수 t를 계산합니다. **교차 조건**은 $(\mathbf{C}_w + t\mathbf{d}_w)_z = \text{ground_Z}$입니다. 즉,

  $$C_{w,z} + t \cdot (d_{w,z}) = \text{ground\_Z}\,,$$

  여기서 $d_{w,z}$는 d_w의 Z성분입니다. 이 식을 풀면

  $$t = \frac{\text{ground\_Z} - C_{w,z}}{d_{w,z}}$$

  로 계산할 수 있습니다. 각 모서리 광선에 대해 t를 구하고, **교차점 좌표** $P = C_w + t \cdot d_w$를 얻습니다. 이렇게 하면 이미지 4개 코너가 지면에 투사된 **4개의 좌표 (P1, P2, P3, P4)**가 결정됩니다.

- **Footprint 폴리곤 구성**: 얻어진 4개의 교차점 P_i들은 일반적으로 지면 위의 **볼록 사변형**을 이루며, 이것이 해당 이미지의 지상 투영 영역(footprint)이 됩니다. 이 4점으로 정의되는 **convex hull**을 구하여 **footprint 다각형**을 표현합니다. 대부분의 경우 카메라가 지면 위에서 내려다보는 구도이므로, 이미지의 네 구석은 지면에 볼록한 형태로 투영됩니다 (어안 렌즈나 특수한 경우가 아니라면). 따라서 4개 점의 **순서를 올바르게 정렬**하여 **볼록 4변형(polyon)**으로 취급하면 정확한 footprint 영역을 얻을 수 있습니다. 필요에 따라 이 폴리곤을 감싸는 **축 정렬 경계 상자(AABB)**를 계산하여 사용할 수도 있습니다 (예를 들어 연산 간소화를 위해). 그러나 단순히 AABB만 사용할 경우 경사 촬영 등에서 부정확성이 생길 수 있으므로, 가능하면 **폴리곤 자체를 사용**하거나 최소한 해당 폴리곤의 **convex hull**을 이용하는 것이 정확합니다.

- **고도 계산 수정 (중요)**: 위 과정에서 **카메라 중심 Cw의 Z좌표와 ground_Z의 차**를 올바르게 사용해야 합니다. 즉 **카메라 실제 고도 = C_{w,z} - ground_Z**로 산정합니다. 기존 코드에서는 이를 무시하고 Cw의 절대 Z좌표를 그대로 사용했기 때문에 오류가 발생했습니다. 리팩토링된 함수에서는 입력으로 받은 ground_Z 값을 이용해 **상대 고도**를 고려하며, 위 교차 계산에서 **t 값 계산 시 ground_Z를 사용**함으로써 정확한 footprint 크기를 얻습니다. 결과적으로 **footprint 영역이 실제 촬영 지면 범위에 부합**하게 되어, 이후 단계의 필터링 정확도가 높아집니다.

- **반환**: 함수는 계산된 footprint를 **다각형 또는 경계 영역** 형태로 반환합니다. 구체적으로:

- 4개 코너의 월드 좌표 리스트 $[P1, P2, P3, P4]$를 반환하거나,
- 필요에 따라 이들로부터 계산한 **convex hull 다각형**을 반환할 수 있습니다.
- 만약 단순 경계박스가 필요하다면, 4점의 최소/최대 좌표로 **AABB**를 산출하여 반환합니다. 그러나 본 시스템에서는 최대한 정확한 영역을 활용하기 위해 **convex hull 다각형**을 기본 반환값으로 고려합니다. 이로써 **이전의 단순 bbox 근사**보다 **정확한 지면 투영 영역**을 얻을 수 있습니다.

좌표계 정의 및 회전행렬 활용

정확한 계산을 위해 **좌표계의 정의**와 **회전 행렬의 의미(R_c2w vs R_wc)**를 명확히 해야 합니다.

- **월드 좌표계 (World Coordinate System)**: 포인트 클라우드와 카메라의 외부 파라미터(EOP)가 정의된 공통 좌표계입니다. 일반적으로 Pix4D 등 소프트웨어에서 출력한 좌표계로, 이 시스템에서 X, Y는 수평 평면 좌표 (예: 동쪽, 북쪽 방향 축 등), **Z는 수직 높이(고도)**를 나타냅니다. 각 사이트에 대한 **평균 지면 높이 ground_Z**는 `report.xml` 내 `<coordinateSystem><output><xyzMean z="...">`에서 제공되는 값으로, 월드 좌표계에서의 기준 고도 오프셋이라고 볼 수 있습니다. (예: ground_Z=109m이면, 해당 좌표계에서 평균 지면이 Z≈109에 위치한다는 의미이며, 내부적으로 이 값을 사용해 좌표를 평탄화시켰을 수 있습니다.)

- **카메라 좌표계 (Camera Coordinate System)**: 카메라 자체의 좌표계로, **원점이 카메라 중심(Cw)**에 있고, 보통 **Z축이 카메라의 앞방향(광축)**을 가리킵니다. X축, Y축은 카메라 이미지 센서의 수평/수직 방향과 평행이

며, 일반적으로 **우-Handed 좌표계**로 정의됩니다. (예를 들어, 많은 컴퓨터 비전 라이브러리에서는 카메라 좌표계에서 X축은 이미지의 오른쪽 방향, Y축은 이미지의 아래쪽 방향, Z축은 렌즈 앞방향으로 정의됩니다. 이는 이미지 좌표 (u 방향 = +X, v 방향 = +Y)와 일치시키기 위한 설정입니다. 다만 사용된 photogrammetry 소프트웨어에 따라 Y축 방향 정의가 달라질 수 있으므로 내부 모듈의 정의를 따라야 합니다.)

- **회전 행렬 R_c2w vs R_wc**:

- **R_c2w (Camera-to-World)**: 카메라 좌표계의 벡터를 월드 좌표계로 변환하는 3x3 회전행렬입니다. 예를 들어 카메라 좌표계에서 방향 벡터 $\mathbf{d}_c$가 주어지면, $\mathbf{d}_w = R_{c2w}\mathbf{d}_c$로 변환하면 월드 좌표계 기준의 방향 벡터를 얻습니다. 마찬가지로 카메라 좌표의 한 점 $\mathbf{X}_c$를 월드 좌표계로 회전시키려면 $\mathbf{X}_w = R_{c2w}\mathbf{X}_c$로 계산합니다. **주의**: 여기서 R_c2w는 순수 회전만 취급하며, 위치 이동은 별도로 카메라 중심 Cw를 더해야 완전한 변환이 됩니다.
- **R_wc (World-to-Camera)**: 월드 좌표계의 벡터를 카메라 좌표계로 변환하는 회전행렬입니다. 이는 R_c2w의 **전치행렬(transpose)** 혹은 **역행렬**에 해당하며, $R_{wc} = (R_{c2w})^{-1} = (R_{c2w})^T$입니다 (회전행렬은 직교행렬이므로 역이 곧 전치입니다). 일반적으로 외부 표정 요소(EOP)를 통해 얻어지는 회전값은 카메라의 자세를 나타내는데, 문맥에 따라 R_c2w 또는 R_wc로 제공될 수 있습니다. **Pix4D 등의 보정 결과**에서는 흔히 R_wc (월드 -> 카메라)가 제공되고, 카메라 중심 Cw (월드 좌표계)가 따로 주어집니다. 본 리팩토링에서는 입력으로 **R_c2w를 직접 받도록 설계**하였으므로, 만약 기존 데이터가 R_wc 형태라면 이를 전치하여 사용해야 합니다.

- **검증**: R_c2w의 정의를 검증하려면, 예를 들어 카메라 좌표계의 단위 벡터 $(0, 0, 1)$ (카메라 정면)을 R_c2w로 변환했을 때 월드 좌표계에서 **카메라가 바라보는 방향 벡터**가 올바르게 나오는지 확인하면 됩니다. 마찬가지로 $(0, 1, 0)$ (카메라 좌표계 위쪽)이 월드 좌표계에서 어떤 방향이 되는지도 확인하여, 카메라의 roll/pitch/yaw 적용이 예상대로인지 검증할 수 있습니다.

- **좌표계 일관성**: `camera_io`, `camera_calibration` 모듈 등에서는 위 좌표계와 행렬에 대한 정의를 이미 사용하고 있을 것입니다. 리팩토링시 이들과 **일관성을 유지**해야 합니다. 예를 들어, `camera_calibration` 모듈에서 얻은 **내부파라미터 (fx, fy, cx, cy)**는 이미지 좌표 -> 카메라 좌표 변환에 일관되게 사용되어야 합니다. 또한 `camera_io`에서 불러온 **카메라 자세 (R, 위치)**가 R_c2w로 해석되는지 확인해야 하며, 필요시 변환해주어야 합니다. 이번 수정에서는 가능하면 모듈의 기존 인터페이스를 변경하지 않고, `compute_camera_footprint()`를 해당 모듈 또는 적절한 위치에 구현하여 다른 부분 (예: forward 픽셀 처리 루틴)에서 사용하도록 합니다.

Forward 처리 파이프라인 수정

`part3_forward_pixelwise.py` 등 Forward 투표(전방 투영) 알고리즘의 파이프라인에서는 **footprint 기반 필터링**을 수행합니다. 기존에는 잘못 계산된 footprint (모든 포인트를 덮는 bbox)로 인해 필터링 로직이 매번 "이미지 영역 내 점 수 > 임계값"으로 판정되어 GPU 연산을 생략하고 있었습니다. 이를 개선하기 위한 파이프라인 수정 사항은 다음과 같습니다:

1. **Footprint 정확 계산 적용**: 각 입력 이미지마다 새로 구현한 `compute_camera_footprint()`를 호출하여 **정확한 지상 투영 다각형**을 계산합니다. 이전에는 단순히 **카메라 높이와 FOV로 bbox를 근사**했다면, 이제는 **4-코너 광선 추적**을 통해 얻은 **폴리곤 영역**을 사용합니다. 이 폴리곤을 이용해 해당 이미지가 **실제로 커버하는 지면 영역**을 정확히 파악합니다.

2. **포인트 클라우드 필터링**: 계산된 footprint 폴리곤에 대해, **포인트 클라우드 내 점들을 영역 필터링**합니다. 구체적으로, 포인트 클라우드의 각 포인트가 이 폴리곤 내부에 속하는지 테스트하여, **해당 이미지가 덮는 영역의 점들만 추출**합니다. 이 과정은 예를 들어 2D 평면 상의 point-in-polygon 테스트나, 사전에 포인트 클라우드를 2D 그리드/R-tree 등으로 공간 색인화하여 빠르게 수행할 수 있습니다. 필터링 결과로 얻은 포인트들의 수를

N_{pts}라고 하면, 이후 GPU로 전달할 데이터의 범위가 이 집합으로 한정됩니다. **이 필터링 덕분에** 각 이미지 처리시 불필요한 연산(이미지가 보지 못하는 영역의 포인트 처리)을 줄이고 메모리 사용을 최적화합니다.

3. **Z-필터링 옵션화**: 기존 구현에서 **포인트의 Z값 기반 추가 필터링**이 있었다면, 이를 **옵션**으로 조정합니다. (예: 카메라보다 높은 곳에 있는 점이나, 지면에서 너무 벗어난 점을 제외하는 등의 로직이 있었을 수 있습니다.) 요청사항에 따라, **포인트 Z 필터링을 끌 수 있는 옵션**을 추가합니다. 즉, 경우에 따라 **지면 영역 내의 모든 점**을 사용하도록 하거나, 또는 필요 시 **Z 범위 조건**을 적용하도록 합니다. 기본 동작은 **Z 필터링 미적용** (즉, footprint 수평범위만 고려)으로 하고, 설정에 따라 **높이 조건 (예: 카메라 고도보다 높은 점 제외 등)**을 선택적으로 적용할 수 있게 합니다. 이 변경으로 사용자는 특정 시나리오(예: 지형 기복이 크지 않거나 모든 포인트를 고려해야 하는 경우)에 **모든 포인트를 활용**할 수 있고, 필요 시 **노이즈나 비관심 높이대 영역을 필터링**할 수 있습니다.

4. **GPU 메모리 관리 및 동적 처리**: 필터링 후 선택된 포인트 수 N_{pts}에 따라 GPU로의 **처리 여부를 결정**합니다. 과거에는 임의로 정한 임계값 (예: 10백만 포인트, 10M) 이상이면 메모리 초과를 우려해 **처리를 생략(SKIP)**하는 로직이 있었습니다. 그러나 이런 고정 임계값은 환경에 따라 너무 보수적일 수 있으므로, 이번 개선에서는 **동적으로 GPU 메모리 예측 및 임계 판단**을 하도록 수정합니다.

5. **GPU 메모리 사용량 예측**: N_{pts}과 이미지 해상도 등을 기반으로, 해당 연산에 필요한 버퍼 메모리를 추산합니다. 예를 들어, 포인트당 처리에 필요한 데이터 (포인트 좌표, 색상 등)와, Z-버퍼 등 **GPU상 필요한 배열 크기**를 계산하여 메모리 요구량을 산출할 수 있습니다. 이때 시스템의 GPU 총 메모리와 현재 가용 메모리를 조회하여 **여유 공간 대비 요구량**을 비교합니다.
6. **임계값 완화 및 동적 조정**: 기본 10M 포인트 제한을 **유연하게 조정**합니다. 예를 들어, 만약 N_{pts}이 12M이라도 GPU 메모리가 충분하다면 그대로 처리하도록 허용하고, 반대로 8M일지라도 메모리 여유가 부족하면 경고를 주는 식입니다. 또한 임계값을 설정값으로 두어 **사용자가 조정**할 수 있게 합니다 (예: 설정 파일이나 인자로 1000만, 2000만 등 변경 가능). 최종적으로, **footprint 내 포인트 수가 너무 많아 한 번에 처리하기 어려울 경우**, **여러 차례에 나눠서 처리(batching)**하거나 **해당 이미지를 건너뛸지**를 결정합니다.

7. **GPU 연산 진행/스킵 조건**: 새로운 기준에서는 "N_{pts}가 임계값을 초과하면 바로 SKIP" 대신, **여러 조건을 검토**합니다. 우선 N_{pts} 대비 메모리 여유, 그리고 한번 처리에 소요되는 예상 시간 등을 종합 고려합니다. 만약 N_{pts}가 다소 많더라도 GPU에 충분한 메모리가 있고, 처리 시간도 수용 가능하면 **그대로 처리**합니다. 반면 정말 과도하게 큰 경우에는 경고를 출력하거나 **자동으로 batch 분할 처리**를 시도합니다. 이로써 **과도한 보수적 스킵**을 줄이고, **GPU 리소스를 최대한 활용**하여 작업 효율을 높입니다.

8. **Forward 투표 연산 진행**: 위의 필터링 및 조건 검토를 통과하면, 해당 이미지에 대해서 **GPU 기반의 Z-버퍼 forward 프로젝션 연산**을 수행합니다. 이 연산은 선택된 포인트들을 카메라 뷰로 **렌더링/투영**하여 Z-버퍼를 생성하고, 각 포인트에 대해 **보텀업(Bottom-up) 방식**으로 픽셀 가중 투표를 하거나, 혹은 여러 이미지 간 **겹침 영역에서의 투표/블렌딩**을 수행하는 것으로 이해됩니다. (구체적인 투표 알고리즘은 기존 `part3_forward_pixelwise.py` 에 구현되어 있을 것입니다.) 개선된 footprint 적용으로, 연산은 이제 **실제로 겹치는 부분에 대해서만** 이뤄지므로, **GPU 부하가 감소**하고 **효율은 향상**됩니다. 또한 Z-버퍼 비교 등에서 사용할 포인트 집합이 줄어들어, **픽셀 단위 연산의 충돌 가능성도 줄고 정확도는 높아질 것**입니다.

9. **반복 및 결과 취합**: 모든 이미지에 대해 위 절차 (footprint 계산 -> 포인트 필터링 -> GPU 투영/투표)을 수행합니다. 처리된 결과(예: 각 포인트에 대한 최적 영상 선택 또는 색상 투표 결과 등)를 모아서, 필요한 최종 산출(예: 텍스처 생성, 가시도 계산 등)을 얻습니다. 이전과 달리, 이제는 **이미지별로 GPU 연산이 스킵되는 일이 거의 없으며**, **필요한 부분만 효율적으로 처리**하게 됩니다.

10. **로깅 및 검증**: 수정 사항을 적용한 후, 각 단계 (특히 필터링 결과 포인트 수, GPU 메모리 판단 부분)에 대한 **로그를 남겨** 사용자가 확인할 수 있도록 합니다. 예를 들어, 각 이미지마다 "Footprint polygon = [...], Points in range = X, GPU processing = proceeded/skipped" 등의 메시지를 출력하거나 기록하여, 시스템 동작을 투명하게 파악할 수 있게 합니다. 이는 추후 튜닝이나 문제 발생시 진단에 도움이 됩니다.

GPU 연산량 및 임계값 조정

GPU 메모리 관리와 연산 부하는 본 시스템의 성능과 안정성에 매우 중요한 요소입니다. 이번 개선에서는 다음과 같이 **GPU 연산량 추정과 임계값 관리 전략**을 명시합니다:

- **GPU 연산량 추정**: 한 장의 이미지에 대해 forward 투영을 실행할 때 필요한 연산량은 대략 **O(N_{\text{pts}})**에 비례하며, 여기서 N_{pts}는 해당 이미지 footprint 내 포함된 포인트 수입니다. 또한 **메모리 사용량**은 포인트 수에 비례하여 증가하고, 이미지 해상도에 따라 Z-버퍼 및 결과 저장을 위한 메모리가 추가로 필요합니다. 예를 들어, $N_{\text{pts}} = 5$백만이고, 포인트당 32바이트의 데이터 구조(좌표, 색상 등)를 사용한다면 포인트 데이터에만 약 160MB가 필요합니다. 여기에 Z-버퍼나 부가 버퍼 (예: 8-bit depth map이 이미지 해상도로 할당) 등이 추가되면 수십 MB가 더 요구될 수 있습니다. 이처럼 **입력 규모에 따른 메모리 요구를 사전에 계산**하여, 현재 GPU 여건에서 감당 가능한지 판단합니다.

- **임계값(Threshold) 설정**: 기존의 고정 임계값 **10M 포인트**는 일부 상황에서 지나치게 보수적일 수 있습니다. GPU 메모리가 충분한 최신 하드웨어에서는 10M 이상도 처리 가능하며, 반대로 메모리 적은 환경에서는 10M 이하도 위험할 수 있습니다. 따라서 임계값을 **유연하게 조정**합니다.

- 기본 임계값을 예를 들어 **20M** 등으로 상향 조정하여 여유를 두되, **실제 판단은 동적 메모리 상황에 기반**합니다.
- 또한 설정 파일이나 커맨드 라인 인자를 통해 **사용자가 직접 임계값을 설정**할 수 있도록 합니다. 이를 통해 운영자는 자신의 GPU 메모리 (예: 6GB vs 24GB)에 맞춰 보수적이거나 적극적인 값을 선택할 수 있습니다.

- 코드 상에서는 N_{pts}가 임계값을 넘는 경우에 대해 **분기 처리**를 합니다. 단순 SKIP이 아니라, "**경고 출력 후 처리 강행**", "**처리 범위를 쪼개어 나눠 처리**", "**연산 모드를 변경**(예: 저해상도 처리)" 등의 대안을 고려합니다. 이번 개선 범위에서는 **batch 분할 처리**와 **SKIP 최소화**에 중점을 둡니다.

- **동적 메모리 체크**: 실행 시점에 **GPU의 전체 메모리와 사용 중 메모리**를 쿼리하여 (CUDA 혹은 OpenGL API 이용), **사용 가능한 남은 메모리 양**을 얻습니다. 그리고 추정된 필요 메모리와 비교하여, 만약 여유가 충분하면 임계값을 넘더라도 **일시적으로 허용**하거나, 여유가 부족하면 임계값 이하라도 **경고**를 주는 식으로 동작합니다. 예를 들어, 임계값 15M으로 설정되었지만 현재 GPU에 여유 메모리가 매우 많다면 18M짜리도 시도해보고, 반대로 10M이어도 이미 다른 작업으로 메모리가 대부분 찼으면 안전을 위해 SKIP하거나 대기하도록 하는 식입니다. 이러한 **실시간 메모리 인지형 처리**는 시스템의 안정성을 높여주며, 자원을 최대로 활용하게 해줍니다.

- **임계값 초과 시 대처**: 만일 필터링 후의 포인트가 너무 많아 (예: 30M 이상 등) 한 번에 처리하기 어렵다면, 아래와 같은 전략을 취합니다:

- **Batching**: 해당 이미지의 footprint를 적절히 분할하여 (예: 폴리곤 영역을 2등분 혹은 4등분하는 등) 각 부분에 대해 포인트를 나눠 처리하고, 결과를 합칩니다. 또는 포인트 목록을 임의로 반씩 나눠 두 차례에 걸쳐 Z-버퍼 투영을 수행한 뒤 결과를 합성할 수 있습니다. Z-버퍼 기반 투영의 특성상 동일 픽셀 영역에 두 번 나눠서 그려도 병합 로직만 제대로 처리하면 일관된 결과를 얻을 수 있습니다.
- **해상도 축소**: 필요시 GPU 부하를 줄이기 위해 이미지 해상도를 낮춰서 처리하는 옵션도 생각해볼 수 있습니다 (예: 이미지/버퍼를 절반 해상도로 다운샘플링하여 투영, 투표 후 결과를 업샘플링). 다만 이 방법은 투표 정확도에 영향을 줄 수 있으므로 신중히 적용해야 합니다.
- **최후 수단 SKIP**: 만약 어떤 이미지의 처리 비용이 지나치게 커서 위 대안으로도 어렵다면, **해당 이미지를 건너뛰는 것도 최후의 선택지**로 남겨둡니다. 이때는 로그에 명확히 해당 사유를 남기고, 사용자에게 이를 보고하여 추가적인 하드웨어 확보나 파라미터 조정이 필요함을 알립니다.

요약하면, GPU 메모리 및 연산 관리 측면에서 **정적이고 획일적인 임계값** 대신, **정확한 추정과 동적 조정**을 도입함으로써, **시스템의 안정성**을 지키면서도 **성능을 최대화**하도록 수정합니다.

결론 및 기대 효과

이번 기술 개선을 통해 UAV Z-버퍼 Forward 투표 시스템의 핵심 문제였던 **잘못된 footprint 계산으로 인한 GPU 연산 스킵** 현상을 해결하였습니다. `compute_camera_footprint()`의 리팩토링으로 **카메라 고도를 정확히 반영한 지상 투영 영역**을 산출하게 되었고, 이 정보를 기반으로 포인트 클라우드를 효율적으로 필터링하여 **필요한 부분만 GPU에서 처리**하도록 파이프라인을 개선했습니다. 또한 **Z-필터링 옵션**과 **GPU 메모리 임계값 동적 관리**를 도입하여, 시스템이 다양한 환경에서도 유연하고 안정적으로 동작할 수 있게 하였습니다.

정리하면, 수정된 모듈들은 **camera_io**, **camera_calibration** 등 기존 구성과 잘 연동되며, Forward 투표의 전체 흐름은 다음과 같은 순서도로 이해할 수 있습니다:

1. **데이터 로드**: 포인트 클라우드 및 각 이미지의 내부/외부 파라미터(EOP, 캘리브레이션) 로드, 사이트 평균 지면 고도 ground_Z 파싱.
2. **footprint 계산**: 각 이미지에 대해 `compute_camera_footprint(fx, fy, cx, cy, width, height, R_c2w, Cw, ground_Z)` 호출 → footprint 폴리곤 획득.
3. **포인트 필터링**: 포인트 클라우드에서 footprint 폴리곤 내부의 점들 추출 (옵션에 따라 Z 조건 추가 적용 가능).
4. **포인트 수 및 메모리 판단**: 추출된 점 개수 N_pts 계산 → GPU 메모리 요구량 예측 → 임계값/여유 메모리 확인.
5. **GPU 처리 여부 결정**: N_pts가 많으면 batch 분할 등 고려, 가능하면 **GPU 처리 진행**, 불가피하면 **SKIP** (로그 출력).
6. **Z-버퍼 투영 및 투표**: GPU 셰이더/커널을 이용하여 선택된 점들을 영상 평면으로 투영, Z-버퍼 비교 및 투표 연산 수행. 결과로 해당 이미지에 대한 투표 기여도(또는 텍스처 값 등)를 계산.
7. **반복**: 모든 이미지에 대해 2~6단계 반복 수행.
8. **결과 통합**: 각 이미지로부터 얻은 투표/가중치 결과를 통합하여 최종 산출물 생성 (예: 텍스처가 입혀진 모델, 혹은 포인트별 최적 영상 매핑 등).
9. **종료 및 보고**: 처리 요약(몇 개 이미지 처리, 스킵 여부, 메모리 사용) 등을 로그/리포트.

개선된 시스템은 **footprint 기반으로 정확하고 효율적인 전방 투영**을 보장하며, 불필요한 GPU 연산을 방지하면서도 실제 필요한 연산은 놓치지 않도록 설계되었습니다. 이를 통해 **GPU 메모리 오류 없이** 안정적인 동작을 기대할 수 있고, 전체 파이프라인의 **처리 속도와 정확도 향상**이라는 효과를 얻을 수 있을 것입니다.

---

[1] Square cropped image equivalent field of view question - Page 2 - PentaxForums.com
https://www.pentaxforums.com/forums/10-pentax-slr-lens-discussion/471105-square-cropped-image-equivalent-field-view-question-2.html