C++과 OpenCV를 통한 최소기반 3D 렌더러의 최적화된 구현 시도

3D Renderer implemented with C++ and OpenCV in Minimal Basis

일저자

소속(국문): prectal123@github

소속(영문): prectal123@github

E-mail : hajunchang501@gmail.com

C++과 OpenCV를 통한 최소기반 3D 렌더러의 최적화된 구현 시도

국문초록

2D 플랫포머 게임 Animal Well의 독립적인 물리/유체/게임시스템 엔진 구현을 통해 감명을 받아, 개인적으로 고민과 시도, 실수와 도전으로 진행하게된 3D 렌더러 구현에 대한 일지의 형식을 갖춘 논문입니다. 본 개발 과정에서는 3D 엔진의 가장 기초적인 부분에서 시작하며 독자적으로 고안한 수학적 모델의 구현과 최적화 기법에 대한 고민에 집중하기 위해 최소한의 라이브러리 및 기존의 개발된 도구를 사용했으며, 대표적으로는 C++의 기본 라이브러리, 행렬 연산을 위한 Eigen 라이브러리, 윈도우 기반 화면으로의 출력을 위한 OpenCV 라이브러리입니다. 3차원의, 세계를 표현하는 가장 단순한 수학적, 기계적 데이터 스키마에서 시작되어 인간이 눈으로 받아들일 수 있는 시각적 표현에 이르기까지의 과정과, 그를 가장 효과적이고 최적화된 방식으로 구현할 방법에 대해 고민합니다.

주제어

OpenCV, C++, 3D 렌더러

1. 개요

Billy Basso의 1인 개발작 Animal Well은 독특한 시스템과 그보다 더 독특한 그래픽 특징을 가집니다. 조악하기 그지없어 보이는 2D 도트 그래픽에서 게임을 진행하다 보면 생각보다 복잡한 물리엔진과 경악스러울 정도로 정교한 유체 시뮬레이션 엔진을 가지고 있었습니다. 자세히 들여다보니, 평범함을 거부한 이 개발자의 손에 의해서 완전히 새로 만들어진 엔진과 그래픽, 게임 로직들로 이 게임은 기존의 Unity나 Unreal Engine과 같은 지극히 당연하고 상업적인 기준에서 벗어나 있는, 최근의 게임 개발 풍토와 거리를 두는 모습을 지녔습니다. 군 생활 도중, 이 게임을 발견하고 이러한 직접 만들어진 게임 엔진에 매력을 느낀 저는 아주 간단한 한 걸음부터 시작해보기로 마음을 먹었습니다. 바로 C++ 언어로 이루어진 3D 그래픽 엔진, 게임 엔진을 만들어 나가는 것입니다. 이 과정은 단순히 널리 알려진 최적화 기법을 가져다 적용하는 구현 연습이 될 수도 있지만 저는 제가 스스로 고민하여 찾아낸 답들과 수학적 모델, 최적화 아이디어들을 적용시켜 저만의 방식으로 만들어 나가는 방향을 선택하고자 합니다.

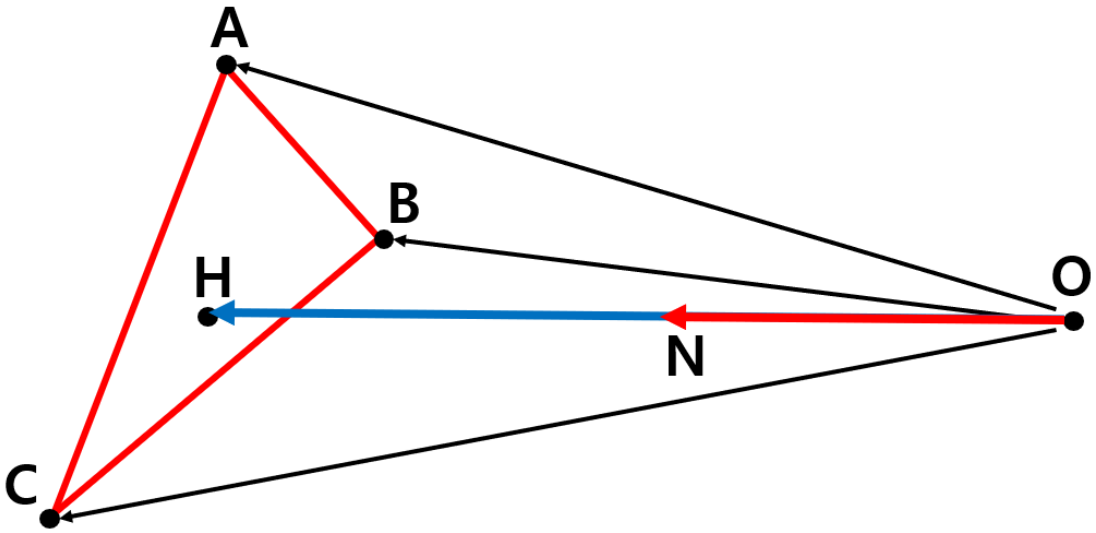
2. 고안 및 구현

2.1. Phase 1 – 스크래치 및 수학적 모델링

2.1.1. 군 생활 중

이 작업에 처음 노력을 기울이기 시작한건 군생활 중 상병의 시기 즈음이었습니다. 그래픽스에 거의 아무런 배경 지식이 없었지만 다양한 경로로 폴리곤에 관한 지식은 있었기 때문에, 가장 단순한 접근부터 시작하고자 했습니다.

하나의 폴리곤이 화면상에 나타나 보이게 하기 위해선 그 폴리곤의 위치를 알아야 했습니다. 3차원 상의 폴리곤, 단순화 시켜 삼각형 하나를 공간상에 나타내 보이게 하기 위해서는 각 Vertex의 좌표를 알고, 현재 카메라 혹은 관찰자의 시점, 시선 방향을 알아야 했습니다. 그래픽 상에서의 각 픽셀에 대해 픽셀 방향의 Ray를 발사한다고 했을 때, 그 Ray가 폴리곤과 충돌하면, 해당 픽셀에는 해당 폴리곤이 렌더링 되어야 한다고 이야기할 수 있을 것입니다. 이 일련의 과정에 대한 판독 원리는 간단 벡터 연산으로 답을 찾을 수 있었습니다.



정점 O를 카메라의 위치, ΔABC를 검사 대상이 될 폴리곤, 그리고 벡터 $\overrightarrow{ON}$를 카메라의 시선 정규 벡터라고 둔다면, $\overrightarrow{ON}$의 연장선인 $\overrightarrow{OH}$ 벡터가 폴리곤과 충돌하는지의 여부를 확인하면 됩니다.

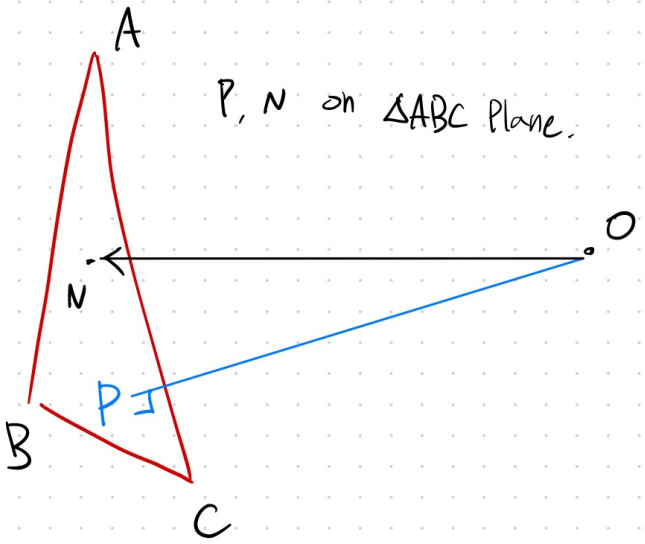
$$\overrightarrow{ON} = a \cdot \overrightarrow{OA} + b \cdot \overrightarrow{OB} + c \cdot \overrightarrow{OC}$$

$$\overrightarrow{OH} = k \cdot \overrightarrow{ON}$$

$$\overrightarrow{OH} = ak \cdot \overrightarrow{OA} + bk \cdot \overrightarrow{OB} + ck \cdot \overrightarrow{OC}$$

이 수식 하에, $\overrightarrow{OH}$ 벡터는 ΔABC의 평면 위에 존재하기 때문에, $ak + bk + ck = 1$이라는 공식이 성립해야 합니다. 물론 첫 공식이 성립하기 위해선 $\overrightarrow{OA}$, $\overrightarrow{OB}$, $\overrightarrow{OC}$가 3차원 상에서의 Basis Vector가 되어야 한다는 조건이 붙지만, 이 조건에 대해서는 추후에 다시 이야기하겠습니다.

여기서 중요한 점은, $\overrightarrow{OH}$ 벡터가 ΔABC의 내부에 안착하기 위해선, ak, bk, ck 모두 양의 값을 가져야 한다는 점입니다. 이 참에 위에서 언급한 수학적 요소를 증명하고 지나가겠습니다. Docx 수식을 사용하기에는 양이 많아 그냥 손으로 썼습니다.



$\overrightarrow{OA} = \overrightarrow{OP} + \overrightarrow{PA}$

$\overrightarrow{OB} = \overrightarrow{OP} + \overrightarrow{PB}$

$\overrightarrow{OC} = \overrightarrow{OP} + \overrightarrow{PC}$

$\overrightarrow{ON} = \overrightarrow{OP} + \overrightarrow{PN}$

i) Is it possible to combine $\overrightarrow{PA}, \overrightarrow{PB}, \overrightarrow{PC}$ to create $\overrightarrow{OP}$?

proof) let $\alpha\cdot\overrightarrow{PA} + \beta\overrightarrow{PB} + \gamma\overrightarrow{PC} = \overrightarrow{OP}$, $|\overrightarrow{OP}| \neq 0$.

$\rightarrow \overrightarrow{OP}\cdot(\alpha\overrightarrow{PA} + \beta\overrightarrow{PB} + \gamma\overrightarrow{PC}) = \overrightarrow{OP}\cdot\overrightarrow{OP} = |\overrightarrow{OP}|^2$

$\rightarrow \alpha\cdot\overrightarrow{OP}\cdot\overrightarrow{PA} + \beta\overrightarrow{OP}\cdot\overrightarrow{PB} + \gamma\overrightarrow{OP}\cdot\overrightarrow{PC} = |\overrightarrow{OP}|^2$

$\rightarrow \alpha\cdot 0 + \beta\cdot 0 + \gamma\cdot 0 = 0 = |\overrightarrow{OP}|^2$ $\rightarrow\leftarrow$ $\therefore$ Not Possible.

---

let $\overrightarrow{ON} = \alpha\cdot\overrightarrow{OA} + \beta\overrightarrow{OB} + \gamma\overrightarrow{OC} = (\alpha+\beta+\gamma)\overrightarrow{OP} + \alpha\overrightarrow{PA} + \beta\overrightarrow{PB} + \gamma\overrightarrow{PC}$,

and as $\overrightarrow{ON} = \overrightarrow{OP} + \overrightarrow{PN}$, $\overrightarrow{OP} + \overrightarrow{PN} = (\alpha+\beta+\gamma)\overrightarrow{OP} + \alpha\overrightarrow{PA} + \beta\overrightarrow{PB} + \gamma\overrightarrow{PC}$.

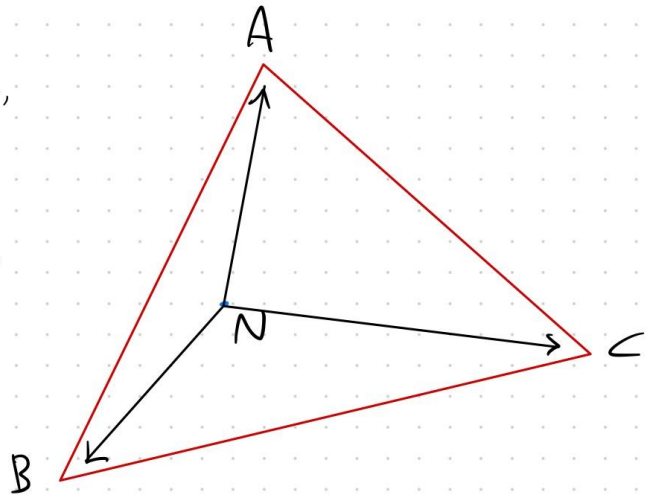
$\rightarrow (1-\alpha-\beta-\gamma)\overrightarrow{OP} = \alpha\overrightarrow{PA} + \beta\overrightarrow{PB} + \gamma\overrightarrow{PC} - \overrightarrow{PN}$,

by proof i), $\overrightarrow{OP}$ cannot be combined by $\overrightarrow{PA}, \overrightarrow{PB}, \overrightarrow{PC}$, or $\overrightarrow{PN}$.

$\therefore (1-\alpha-\beta-\gamma) = 0$, $\therefore \alpha+\beta+\gamma = 1$

ii) Prove if $\alpha, \beta, \gamma > 0$, N is inside $\triangle ABC$.

$\vec{OA} = \vec{ON} + \vec{NA}$,
$\vec{OB} = \vec{ON} + \vec{NB}$
$\vec{OC} = \vec{ON} + \vec{NC}$.



$\therefore$ If $\vec{ON} = \alpha\vec{OA} + \beta\vec{OB} + \gamma\vec{OC} = \alpha\vec{ON} + \alpha\vec{NA} + \beta\vec{ON} + \beta\vec{NB} + \gamma\vec{ON} + \gamma\vec{NC}$

$\because \alpha+\beta+\gamma = 1$, $\vec{ON} = (\alpha+\beta+\gamma)\vec{ON} + \alpha\vec{NA} + \beta\vec{NB} + \gamma\vec{NC}$

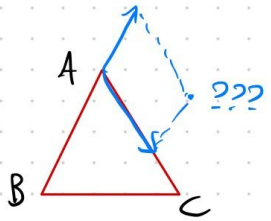
$\therefore \vec{ON} = \vec{ON} + \alpha\vec{NA} + \beta\vec{NB} + \gamma\vec{NC}$.

$\therefore \alpha\vec{NA} + \beta\vec{NB} + \gamma\vec{NC} = 0$.

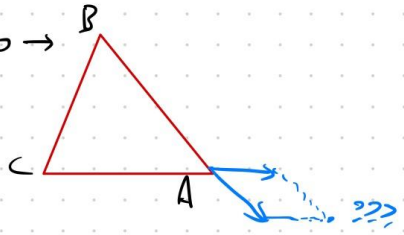
$\rightarrow (1-\beta-\gamma)\vec{NA} + \beta\vec{NB} + \gamma\vec{NC} = 0$, $\rightarrow \vec{NA} + \beta(\vec{NB} - \vec{NA}) + \gamma(\vec{NC} - \vec{NA}) = 0$

$\therefore \vec{NA} + \beta\vec{AB} + \gamma\vec{AC} = 0$. $\therefore \beta\vec{AB} + \gamma\vec{AC} = -\vec{NA}$.

1. WLOG, let $\beta < 0$, $\gamma > 0$. $\rightarrow \beta\vec{AB} + \gamma\vec{AC} = \vec{AN}$



2. WLOG, let $\beta, \gamma < 0 \rightarrow$



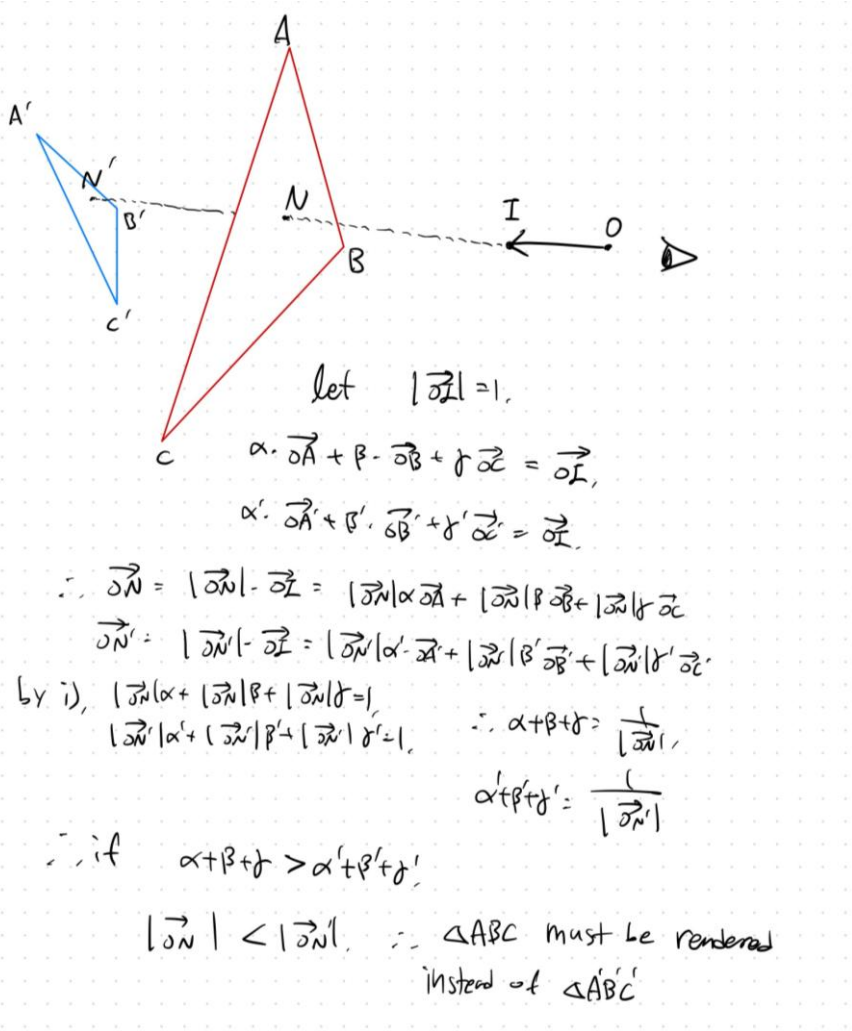
$\therefore \beta, \gamma > 0$.

$\therefore \alpha, \beta, \gamma > 0$.

네. 이제 계수가 각각 합해서 1 이며, 전부 양수여야만 관찰자 시점에서 픽셀 하나의 레이캐스트가 폴리곤 내부에 안착함을 증명했습니다. 완전히 엄밀한 증명은 아니지만, 일단 급한대로 이 프로젝트에 사용할 수 있을 만큼은 수학적 직관을 뒷받침해줄 수 있을 것 같습

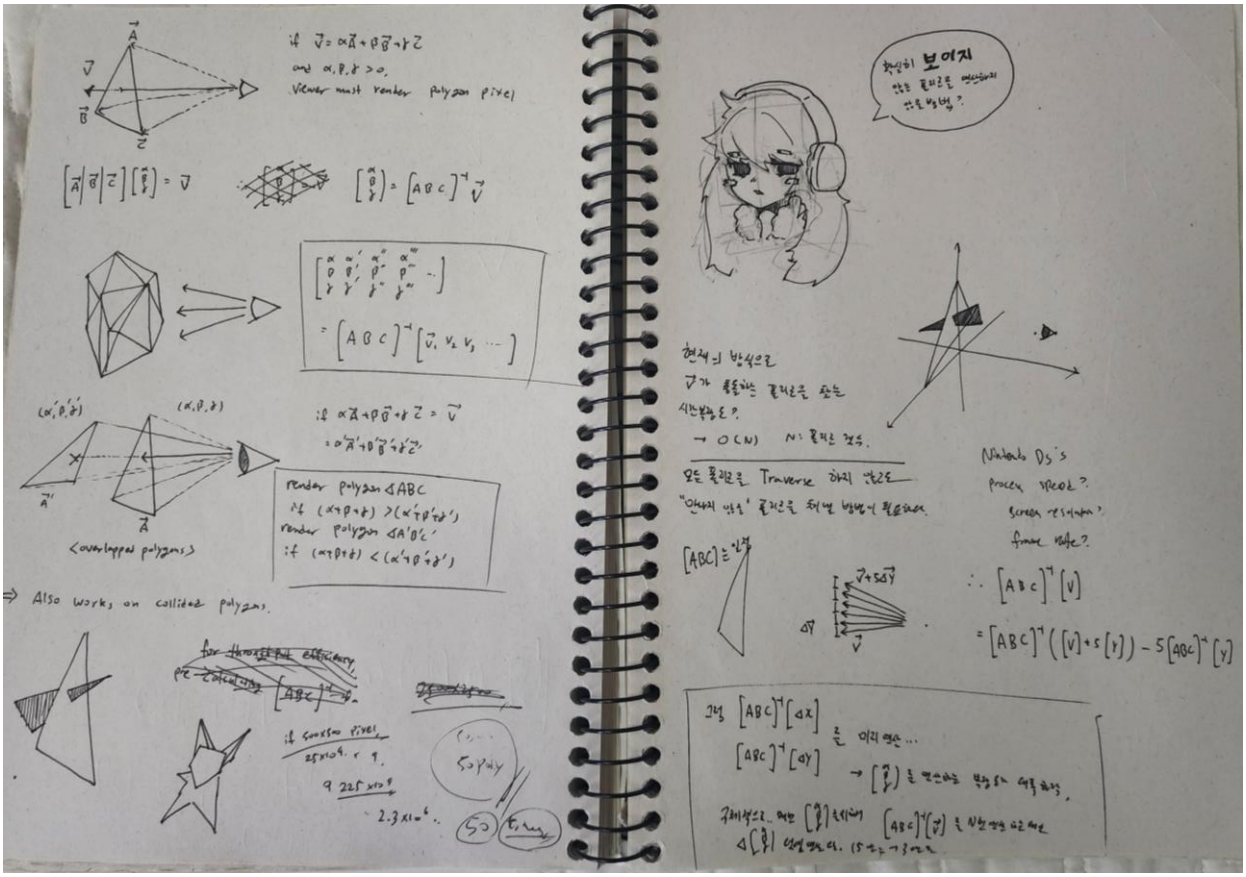
니다. 이제 저희는 하나의 픽셀의 위치, 각도를 알며 대상이 되는 폴리곤의 세 꼭짓점의 위치를 알 때, 이 픽셀에서 해당 폴리곤을 렌더를 해야 할지 말아야 할지를 판정할 수 있게 되었습니다.

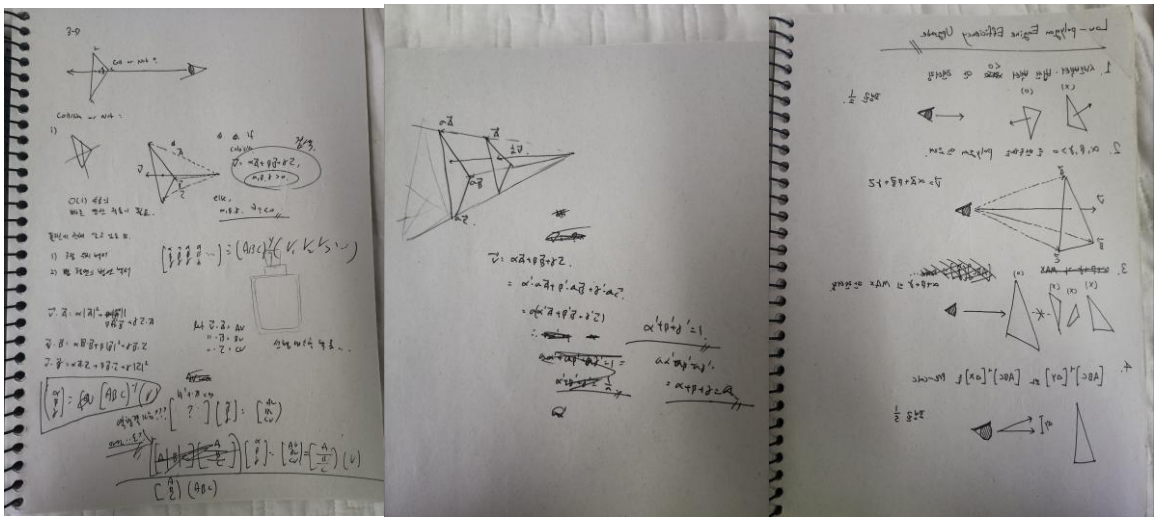
하지만 3D 렌더링에서는 하나의 픽셀이 항상 하나의 폴리곤과 연결되지는 않습니다. 픽셀의 레이캐스트의 연장선상에 여러 개의 폴리곤이 겹쳐 있는 경우는 하나 만의 폴리곤만 있을 때보다 훨씬 많을 것입니다. 그래서 저희는 이제 레이캐스트와 어떤 폴리곤이 가장 먼저 Colide 하는지를 알 수 있어야 합니다. 다행히 이 부분은 단순한 수학적 직관과 증명으로 해결할 수 있었습니다.



뭔가 장황하지만, 요약하자면 카메라의 좌표에서 뻗어나가는 하나의 픽셀의 레이캐스트 상의 단위(혹은 임의의)벡터 $\overrightarrow{OI}$가 존재하고, 두 폴리곤 $\Delta ABC, \Delta A'B'C'$가 있을 때 $\alpha \cdot \overrightarrow{OA} + \beta \cdot \overrightarrow{OB} + \gamma \cdot \overrightarrow{OC} = \alpha' \cdot \overrightarrow{OA'} + \beta' \cdot \overrightarrow{OB'} + \gamma' \cdot \overrightarrow{OC'} = \overrightarrow{OI}$가 성립할 때, $\alpha + \beta + \gamma$가 $\alpha' + \beta' + \gamma'$ 작다면, ΔABC가 $\Delta A'B'C'$를 가리기 때문에, 이를 대신 픽셀에 그리면 되는 것입니다.

군대에 있는 동안에는 머리가 굳었는지 사실 엄밀한 증명까지 가지도 않았습니다. 뭔가 이렇게 되겠거니.... 하면서 구상한게 전부였습니다.





대충대충 끄적이면서 구상한 흔적입니다.

2.2. Phase 2 – 모델 구현

수학적 모델의 구현 과정

2.2.1. OpenGL 대신 OpenCV를 택한 이유

OpenGL은 쉬운 파이프라이닝과 고성능을 추구할 수 있는 가능성이 높아 가장 처음 시도해 본 라이브러리 입니다. 제가 고등학생동안 참여했던 연구 프로그램에서 OpenCV의 사용법을 연습했기 때문에, OpenCV의 사용보다는 무언가 다른 것을 배워보고 싶었습니다. 하지만 제가 이 프로젝트를 시작하면서 다짐했던, 모든 픽셀 하나하나를 수작업으로 찍어내는 단계의 로우레벨에 도달하기에는 OpenGL이 약간 부적합했다고 생각합니다. 이미 그 자체로 3D 모델을 출력해내는 렌더링 기능이 있고, 최대한 로우레벨로 내려가도 2D 공간에 삼각형을 그려내는 기능까지 있으니 제가 원하는 만큼 기초적이지 못했다고 생각합니다. 그래서 결국 저는 Window 생성, 마우스 이벤트 감지, 픽셀 단위 그래픽 표현 기능이 기본적으로 탑재된 OpenCV를 사용하기로 방향을 바꾸었습니다. 사실 OpenCV 마저 불가능했다면 저는 터미널에 아스키코드를 통해 그래픽을 표현하는 방법이라도 사용하려고 했습니다.,

(사실 엄밀히 따지면 OpenGL로도 픽셀 단위 렌더링이 가능하다는 것을 알고 있습니다. 하지만 이 영역은 제 프로젝트의 다음 단계로 삼겠습니다.)

2.2.2. 구현

간단한 튜토리얼을 통해 6년간의 공백을 메우고, OpenCV의 감을 잡은 뒤 본격적인 구현을 시작했습니다. 특정한 공간상의 점들의 x,y,z 축 값을 지정해주고, 카메라의 위치, 카메라의 시선 방향, 카메라의 시선 기준 x축과 y축, 한 픽셀당 할당할 delta x와 delta y를 지정해주고 나서야 비로소 레이캐스트 벡터와 폴리곤의 꼭짓점들의 벡터를 정의할 수 있었고, 위의 공식대로 코드를 만들어 보았습니다.

```
47  + bool detPrint(int index, Vector3d raycast) {
48  +     Matrix3d vertex;
49  +     vertex <<
50  +         vertices[index][0].x() - camPosition.x(), vertices[index][1].x() - camPosition.x(), vertices[index][2].x() - camPosition.x(),
51  +         vertices[index][0].y() - camPosition.y(), vertices[index][1].y() - camPosition.y(), vertices[index][2].y() - camPosition.y(),
52  +         vertices[index][0].z() - camPosition.z(), vertices[index][1].z() - camPosition.z(), vertices[index][2].z() - camPosition.z();
53  +
54  +     Vector3d coefficient = vertex.colPivHouseholderQr().solve(raycast);
55  +     return !(coefficient[0] < 0.0 || coefficient[1] < 0.0 || coefficient[2] < 0.0);
56  + }
```

detPrint() 함수는 가장 간단하게, 대상이 되는 폴리곤의 인덱스와 특정 픽셀에서 뻗어나가는 RayCast 벡터를 받아 이 픽셀에서 이 폴리곤을 그려야 하는지 여부를 판정합니다. 여기에는

재밌는 비밀이 있는데, 저는 처음 이 로직을 고안할 때, 같은 폴리곤에 대해 여러 다발의 RayCast를 Solving함에 있어서 하나하나의 폴리곤에 대해 역행렬을 찾고, 그를 바탕으로 RayCast를 단순히 행렬-벡터 곱연산을 행하면서 성능을 높이고자 하였습니다. 그런데 역행렬 생성 라이브러리를 찾던 도중, 이런 글을 발견했습니다.

Computing inverse and determinant

First of all, make sure that you really want this. While inverse and determinant are fundamental mathematical concepts, in *numerical* linear algebra they are not as useful as in pure mathematics. **Inverse** computations are often advantageously replaced by solve() operations, and the determinant is often *not* a good way of checking if a matrix is invertible.

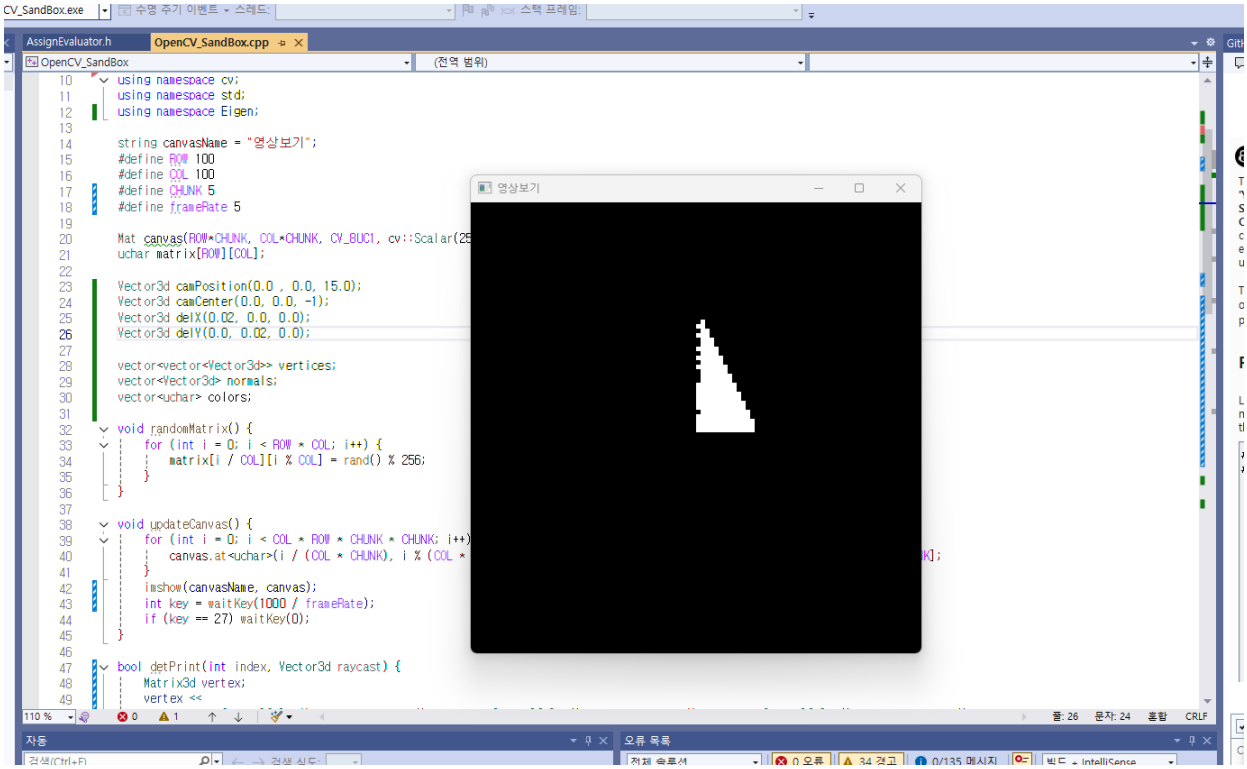
However, for *very small* matrices, the above may not be true, and inverse and determinant can be very useful.

While certain decompositions, such as **PartialPivLU** and **FullPivLU**, offer **inverse()** and determinant() methods, you can also call **inverse()** and determinant() directly on a matrix. If your matrix is of a very small fixed size (at most 4x4) this allows **Eigen** to avoid performing a LU decomposition, and instead use formulas that are more efficient on such small matrices.

[https://libeigen.gitlab.io/eigen/docs-nightly/group__TutorialLinearAlgebra.html]

저에게 보란듯이 역행렬을 함부로 엔지니어링에 포함하지 말고, 큰 행렬 연산에 있어서는 비교적 신뢰도가 높은 Solve() 함수를 사용하는 것이 성능적 이점을 가져오기 쉽다는 이야기였습니다. 실제로 테스트까지 해보진 않았지만, 아직까진 이 이야기를 믿고 있습니다. 시간이 날 때 역행렬 함수를 이용해서도 Throughput을 테스트해보아도 좋을 것 같습니다.

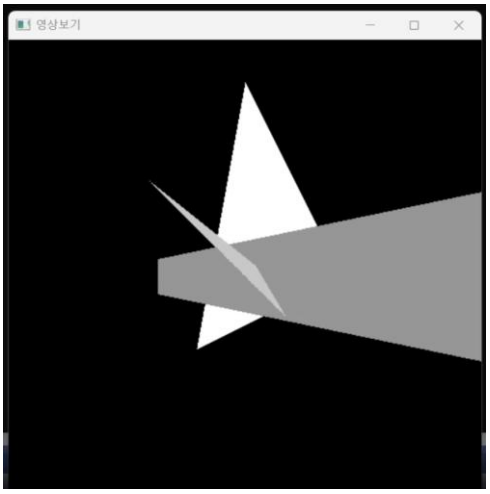
2.2.3. 결과



영광스러운 첫 폴리곤의 모습입니다. 카메라의 위치와 방향, 대략적인 시야각을 설정해주고, 이 정도면 되겠지 싶은 위치와 크기의 폴리곤을 배치했습니다. Floating Point Error로 인해서 왼쪽면의 선이 깔끔하게 이어지지 않고 약간의 노이즈가 있는 모습입니다. 그래도 제가 내었던 공식이 틀리지 않았다는 사실에 기쁜 순간이었습니다.

2.2.4. 고도화 – 더 많은 폴리곤

이제 더 많은 폴리곤을 넣어볼 차례였습니다. 이 부분을 위해서는 폴리곤들의 순서나 RayCast와의 콜리전 순서에 따라 겹쳐 보이거나 가려져 보이는 처리가 필요했습니다. 폴리곤을 3개로 늘리고, 윈도우 크기를 키웠으며, 픽셀 Chunk의 크기를 줄였습니다. 그 결과 다음과 같이 나타났습니다



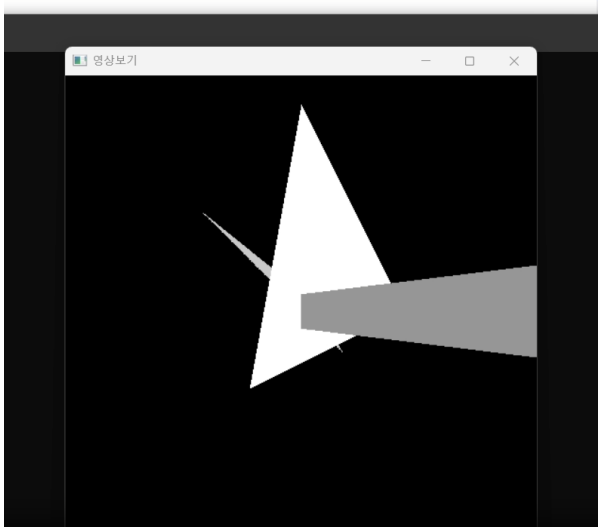
https://github.com/user-attachments/assets/236a1665-f8f4-4c5f-ace1-16d4c09b66c0

얼핏 보면 괜찮아 보이지만, 제대로 작동하지는 않는 중입니다. 이 폴리곤들은 서로 교차되고, 중첩되며 겹쳐져 있어야 하는 상황이었거든요. 현재 각 폴리곤 중 RayCast가 어느 쪽에 먼저 도달하는지 전혀 고려를 안하는 중이었습니다. 그 때문에, 코드를 수정했습니다.

2.2.4.1. 문제 해결 – 폴리곤 RayCast 순서 판단 로직 추가

앞서 언급했던 폴리곤 순서를 결정하기 위해 계수의 합이 가장 큰 폴리곤을 골라 그 폴리곤의 색만을 Canvas에 업데이트 시켜주는 간단한 로직입니다.
https://github.com/prectal123/OpenCV_SandBox/commit/cd13a35dfcd74a639acdbda62aafcd0be61b3439

이를 통해 이제 폴리곤들의 순서를 결정 짓고, 중첩된 모습도 문제없이 처리할 수 있게 되었습니다.



https://github.com/user-attachments/assets/0bd08e10-5323-434e-8c7a-7864068f1fe8

이제서야 제대로 보여집니다. 가장 뒤의 중간 회색 폴리곤은 가려지고, 가장 어두운 폴리곤이 가장 밝은 폴리곤을 제대로 뚫고 나오는 모습입니다.

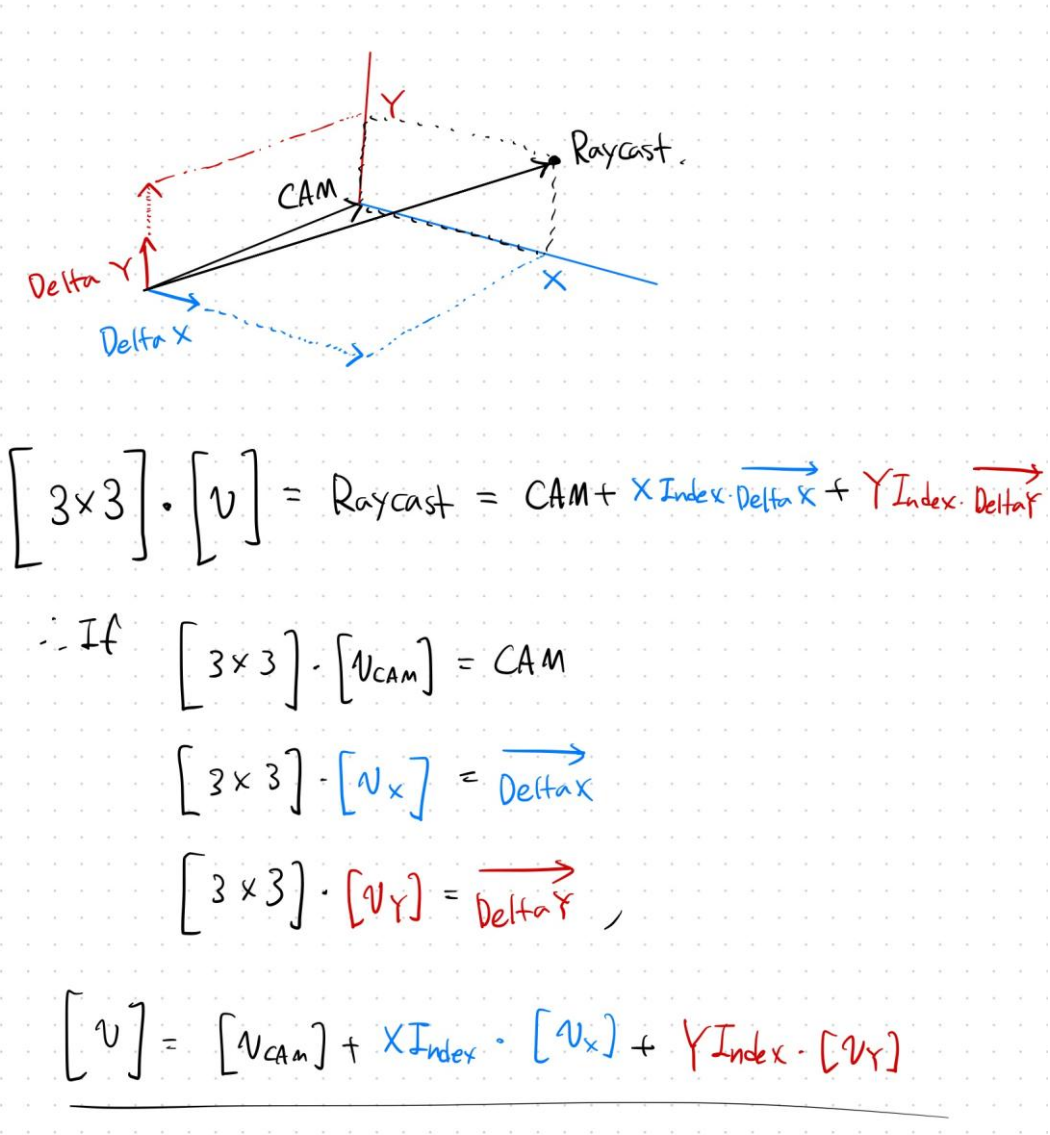
2.2.5. 성능 테스트

그러나 아직 문제가 많았습니다. 위의 링크를 타고 영상을 살펴보면 알 수 있지만, 현재 코드 상 디버깅을 위해 카메라를 천천히 폴리곤의 더미를 향하여 접근하도록 만들어 놓은 상태입니다. 즉, 모든 폴리곤의 렌더링 연산이 끝나고 나면 카메라가 한 걸음씩 나아가는 방식이죠. 그런데 현재, 단 폴리곤이 3개 밖에 없음에도 불구하고 굉장히 느린 성능을 보여줍니다. 잘 쳐줘도 1초에 3프레임 정도 보이는 상황이기에, 유능하고 멋진 3D 엔진과는 동 떨어진 성능을 보여줍니다. 이를 해결하기 위해 어떻게든 현재의 연산 방식을 개선해야 했습니다.

2.2.5.1. 문제 해결 고안 – Solve 연산의 단위화

연산에서 가장 많은 시간을 잡아먹는 장소는 단연 Solve 함수였습니다. 모든 픽셀에서 뻗어나가는 RayCast 벡터에 대해서, 각 모든 폴리곤들의 Vertices로 생기는 3x3 행렬에 대해 Solve를 처리하니, 연산의 숫자가 엄청나게 늘어날 수밖에 없었습니다. 이 때문에 저는 이 연산의 횟수를 줄이는 방법을 고민했고, 선형대수학 강의에서 배웠던 단순한 선형 연산의 특징에서 답을 찾을 수 있었습니다. 현재 제 로직상에서는, 모든 픽셀의 RayCast를 [카메라 시선 벡터] + XIndex * [Delta x 벡터] + YIndex * [Delta y 벡터]와 같은 방식으로 생성해냅니다. 그 말인 즉, 일일이 모든 RayCast들에 대한 Solve 함수를 처리할 필요 없이, [카메라 시선 벡터], [Delta X 벡터], [Delta Y 벡터]에 대한 Solve 처리만 가한 뒤, 그 결과 값들만 선형적으로

조합해주면 수학적으로 동일한 결과물을 O(픽셀 수 * 폴리곤 수)의 시간 복잡도를 O(폴리곤 수)의 시간 복잡도로 비약적으로 줄일 수 있는 것이었습니다. 물론 결과값들을 선형적으로 조합하는 과정에서도 연산이 들긴하지만, Solve 함수의 소요 시간이 길었기 때문에 비약적인 성능 향상을 기대할 수 있었습니다.



2.2.5.2. 문제 해결 – Solve call rate reduce

https://github.com/prectal123/OpenCV_SandBox/commit/15667ff3c9e156a4869cc06f63b6b97a4e9c9b5f

이 로직은 위의 커밋에서 도입되었고, 결과물은 아래의 링크 영상과 같았습니다.

https://github.com/user-attachments/assets/2e1797d9-e76e-47ec-8b29-b603fa0e1d50

비교도 안되게 성능이 개선되었고, 훨씬 부드럽고 빠른 프레임 속도가 나오기 시작했습니다.

3. 참고문헌