한국공학대 게임공학과 2021180007 노훈철

2025-1 3D 게임 프로그래밍 과제 2

과제 설명 문서

과제에 대한 목표 : 과제1을 DirectX12 로 구현하기

실행 결과와 조작법 :

과제1과 동일함.

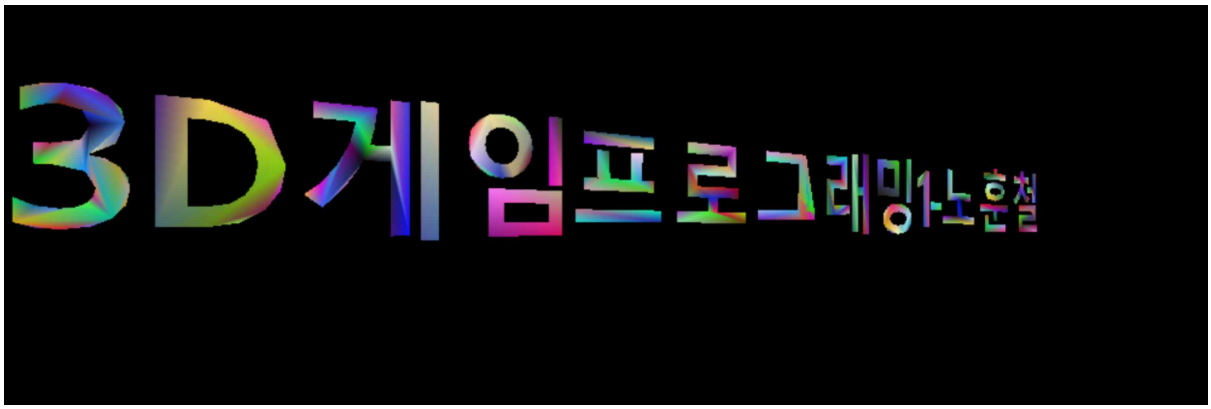
클릭으로 선택.

화살표 키 - 탱크 움직임

W - 강제로 이기기

S - 쉴드 생성

추가 구현내용 - 1. 텍스트 메쉬



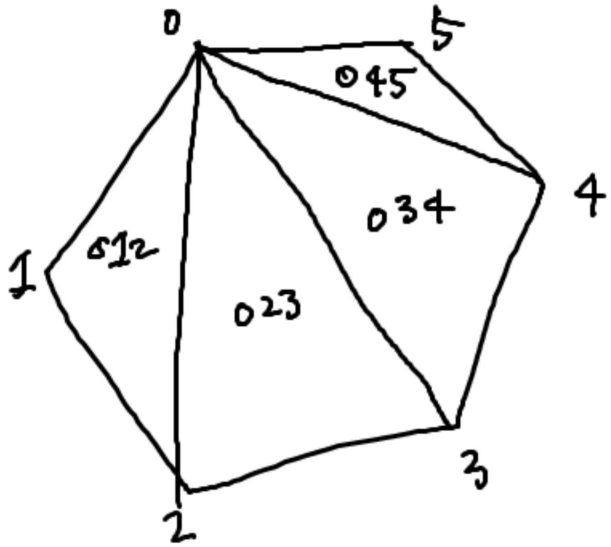
과제 1에서는 베지어 곡선을 활용해 TTF의 외곽선 데이터를 뽑아 게임에 적용했다.

하지만 DirectX12에 그것을 가져오기 위해서 생각보다 더 많은 문제가 있었다.

왜냐하면 모두 삼각형의 메쉬데이터로 글자를 변환했어야 했다. 메쉬데이터라고 하면 결국 삼각형들로 문자를 나타낼 수 있어야 했다. 글자의 모양은 일반적인 다각형이 아니다. 기본적으로 대부분이 볼록한 다각형이 아니다. 그리고 구멍이 뚫린 도형도 있다. 또한 삼각형들은 은면이 제거된다. 때문에 삼각형의 법선 혹은 와인딩 오더는 통일되어야 했다.

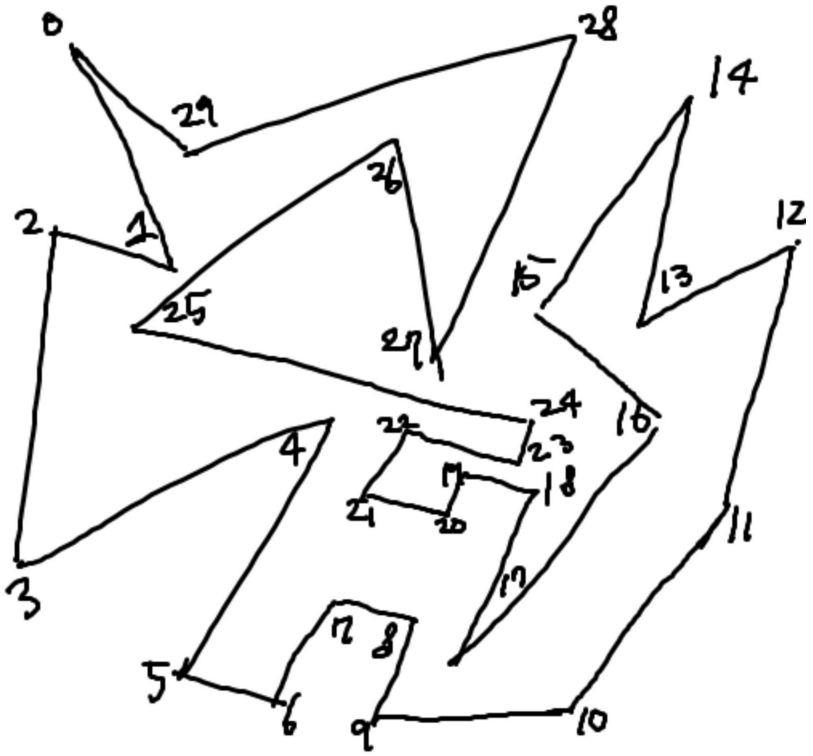
문제**1. 볼록하지 않은 도형을 어떻게 삼각형들의 집합으로 변환할 것인가?**

: 일반적으로 볼록한 도형을 여러개의 삼각형으로 변환하기 위해서는,



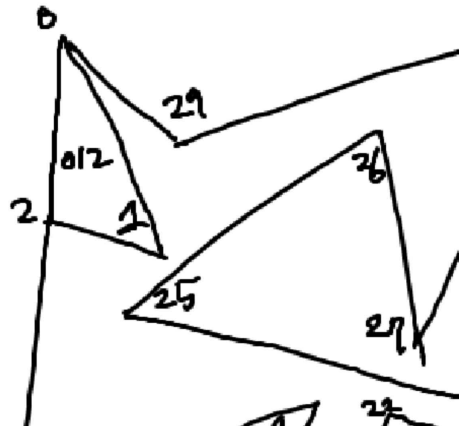
다음과 같이, 0번의 점과, 마지막에 쓰인 점, 그리고 그 다음 점을 연결하면 하나의 삼각형이 만들어지고, 그것을 끝까지 하면 완성된다.

이제 볼록하지 않은 다각형을 한번 보자.



만약 이런 도형의 점들의 데이터가 차례로 주어질때, 이것들을 모두 삼각형들로 바꾸어야 한다면, 어떻게 하는가?

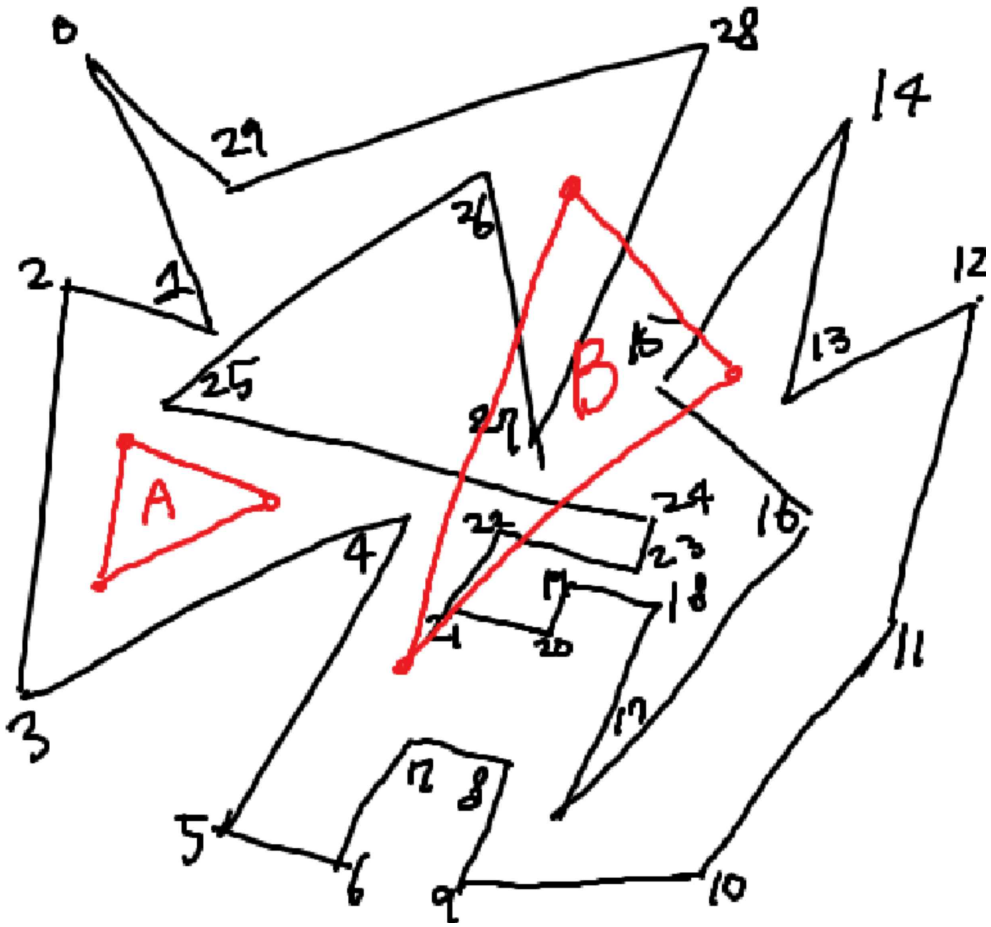
일단 이전과 같은 방식을 고수해보자.



그렇게 되면 012번의 점들이 연결된 삼각형 하나가 나온다. 하지만, 이 삼각형은 이 도형에 포함되어서는 안된다. 왜냐하면 해당 삼각형이 도형의 내부에 존재하지 않기 때문이다.

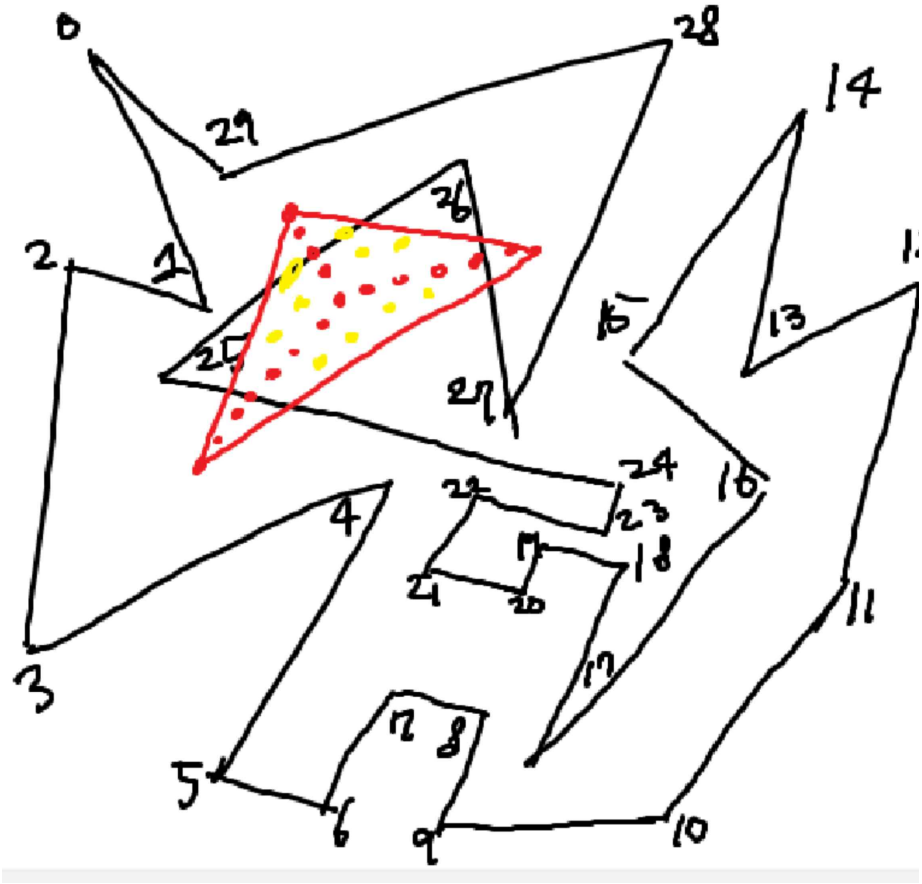
때문에, 삼각형이 폴리곤의 내부에 존재하는지 검사하는 함수 하나를 만들 필요가 있다.

이 함수를 어떻게 만들 수 있을까?



처음에는 삼각형 A와 같이 삼각형을 이루는 세 점이 모두 다각형 내부에 있다면, 삼각형도 내부에 있다고 생각했지만, B와 같은 경우가 나왔다. 삼각형 B의 경우에는 모든 점이 다각형

내부에 있음에도 불구하고, 삼각형이 다각형의 내부에 있다고 말하기는 어렵다.

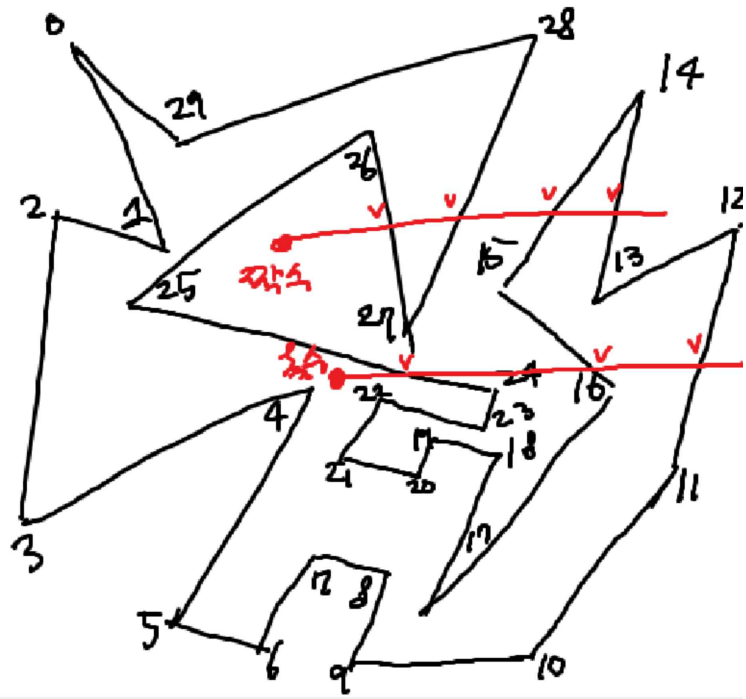


삼각형 내부의 모든 점들이 다각형의 내부에 있을때, 삼각형이 다각형의 내부에 있다. 정확하지는 않더라도, 삼각형 내부의 몇개의 점들을 모두 다각형 내부에 있는지 검사하고, 모두 내부에 있다면, 삼각형 하나가 내부에 있다고 판정한다.

아래 그림은 실제 함수의 모습이다.

```cpp
bool bTriangleInPolygonRange(float t0x, float t0y, float t1x, float t1y, float t2x, float t2y, std::vector<XMFLOAT3> polygon)
{
    XMVECTOR gcenter = XMVectorSet(t0x + t1x + t2x, t0y + t1y + t2y, 0, 0);
    gcenter = gcenter / 3;
    bool result = true;
    if (bPointInPolygonRange(gcenter, polygon) == false) {
        return false;
    }

    XMVECTOR considerP = (gcenter + XMVectorSet(t0x, t0y, 0, 0) * 3) / 4;
    if (bPointInPolygonRange(considerP, polygon) == false) {
        return false;
    }

    considerP = (gcenter + XMVectorSet(t1x, t1y, 0, 0) * 3) / 4;
    if (bPointInPolygonRange(considerP, polygon) == false) {
        return false;
    }

    considerP = (gcenter + XMVectorSet(t2x, t2y, 0, 0) * 3) / 4;
    if (bPointInPolygonRange(considerP, polygon) == false) {
        return false;
    }

    // too many sample to find include triangle.
    for (float a = 0.1f; a < 0.9f; a += 0.1f) {
        float b = 1 - a;

        considerP = a * gcenter + b * XMVectorSet(t0x, t0y, 0, 0);
        if (bPointInPolygonRange(considerP, polygon) == false) {
            return false;
        }

        considerP = a * gcenter + b * XMVectorSet(t1x, t1y, 0, 0);
        if (bPointInPolygonRange(considerP, polygon) == false) {
            return false;
        }

        considerP = a * gcenter + b * XMVectorSet(t2x, t2y, 0, 0);
        if (bPointInPolygonRange(considerP, polygon) == false) {
            return false;
        }

        /* ... */
    }
```

설명을 위해 약간 생략한 부분이 있는데, 한 **점이 다각형 내부에 있는지 판정하는 함수이다.** 삼각형의 내부 판정을 위해서 여러 점들을 내부에 있는지 판정해야 한다. 이것도 그렇게 쉽지는 않다.

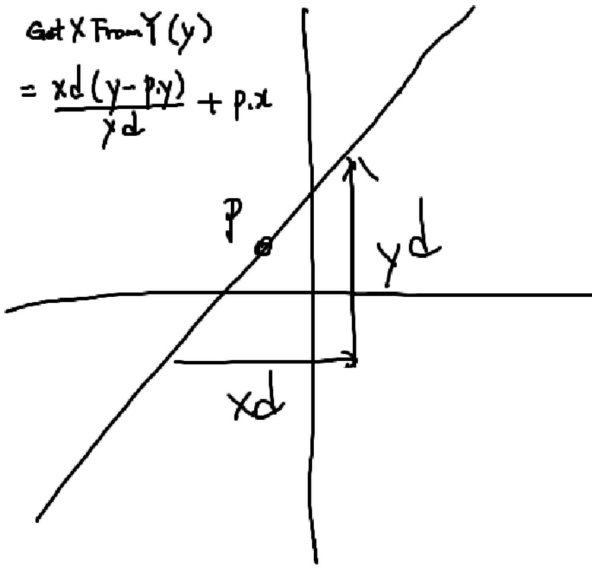


원리는 다음과 같다. 어떤 점에서 반직선을 특정 방향으로 생성한다.

그 반직선과 다각형을 이루는 선들과 교차하는 지점이 짝수면, 외부. 홀수면 내부이다.

이때, 선을 데이터로 표현할때, **그저 직선의 방정식으로 표현하는것은 좋지 않다.**

왜냐하면 y=ax+b 의 경우 표현력이 제한되기 때문이다. x축과 수직인 선은 표현불가하다. 제대로 계산도 안될 것이다. 때문에 선은 {x변화량, y변화량, 지나는 지점} 이를 이용해 선을 대상으로 x를 넣고 y를 받아올 수 있고, y를 넣고 x를 받아올 수 있다.



때문에 점의 내부 판정은 해당 점의 y 값을 기준으로 다각형의 모든 선들에서 x를 얻어오고, 해당 y 값을 가지지 못하는 선들을 제외하고, 해당 점의 오른쪽에 있는 y 값들의 개수가 짝수인지, 홀수인지만 판정한다.

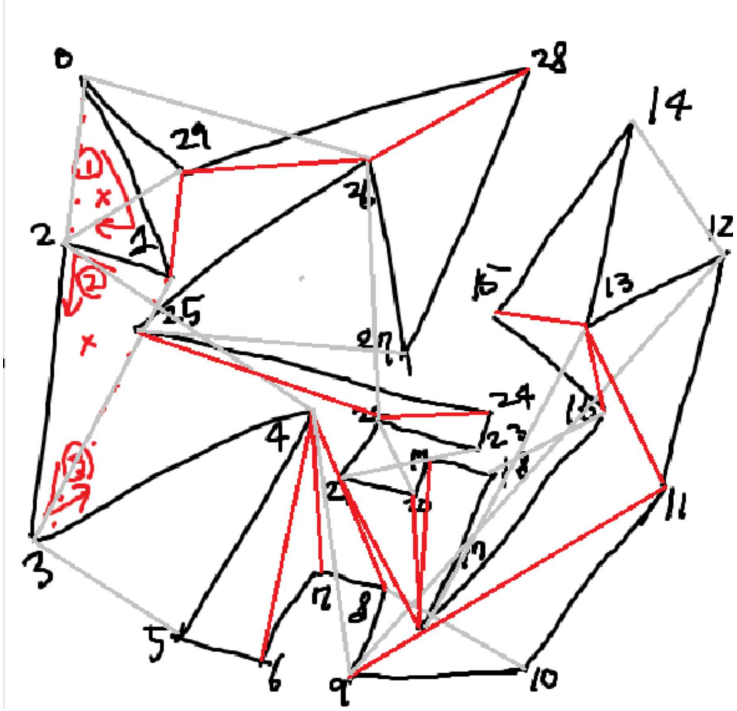
이제 이것들을 이용해 실제로 다각형을 삼각형들로 바꾸어 본다.

아래의 그림의 회색선은, 삼각형 형성에 실패한 시도를 나타내고,

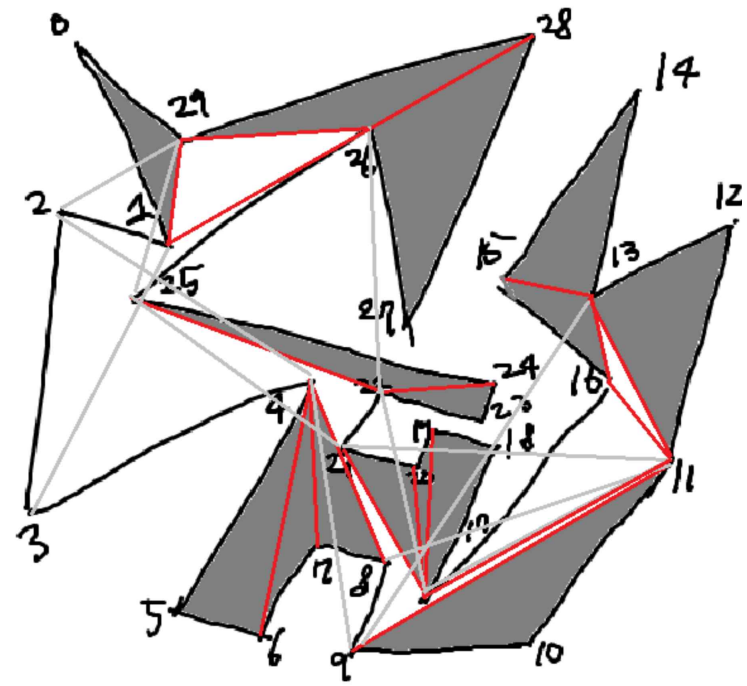
빨간선은 삼각형이 만들어진 시도를 의미한다.

아래 그림은 다각형을 이루는 점들을 한바퀴 돌면서 만들어진 삼각형들과 실패한 시도들을 보여준다.

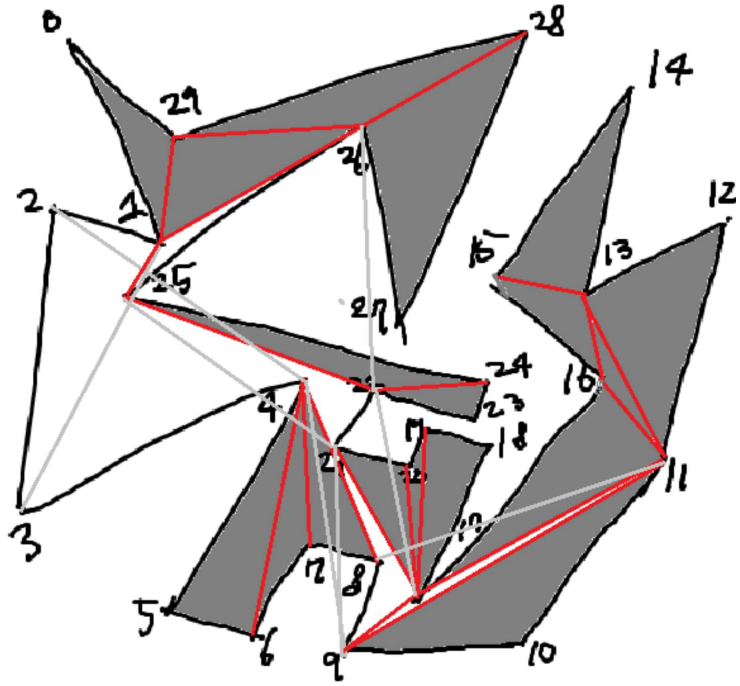
012가 안된다면, 두번째 점을 첫번째 점으로 옮기고, 세번째를 두번째로, 그 다음 점을 세번째 점으로 삼아 다시 삼각형이 만들어질 수 있는지 검사한다.



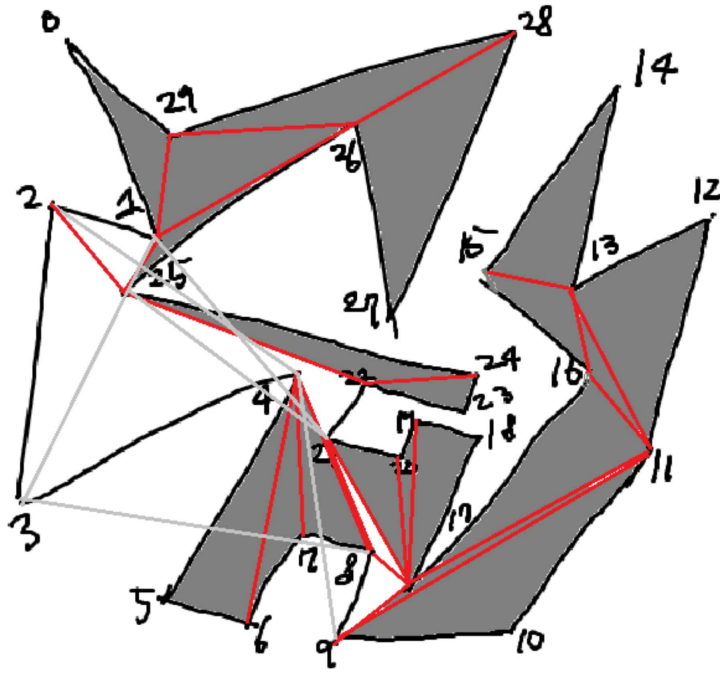
다음은 두번째 루프시 모습이다. 진한 회색은 이미 확정된 삼각형들이다.



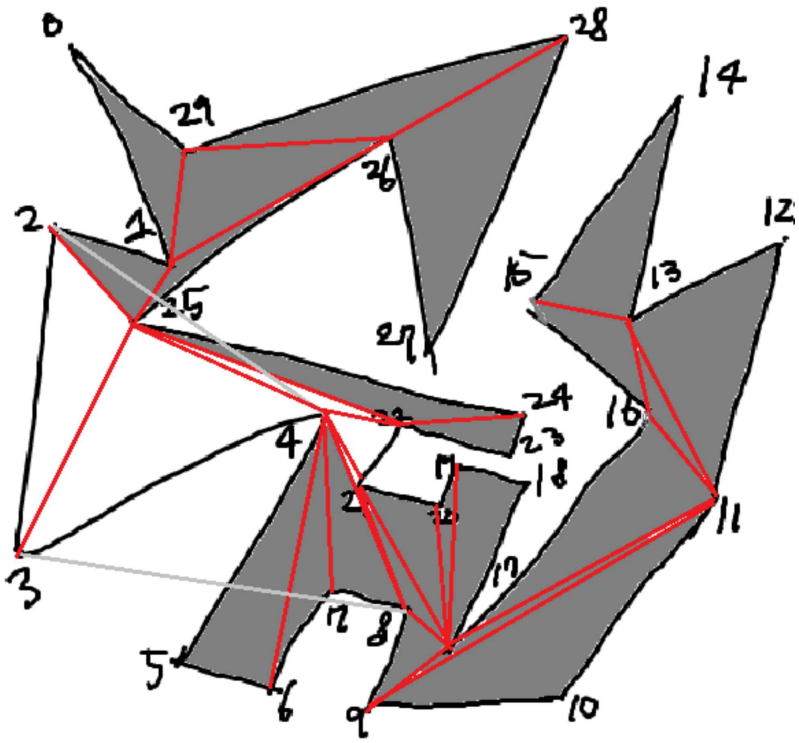
아래 그림은 3번째 루프시 모습이다.



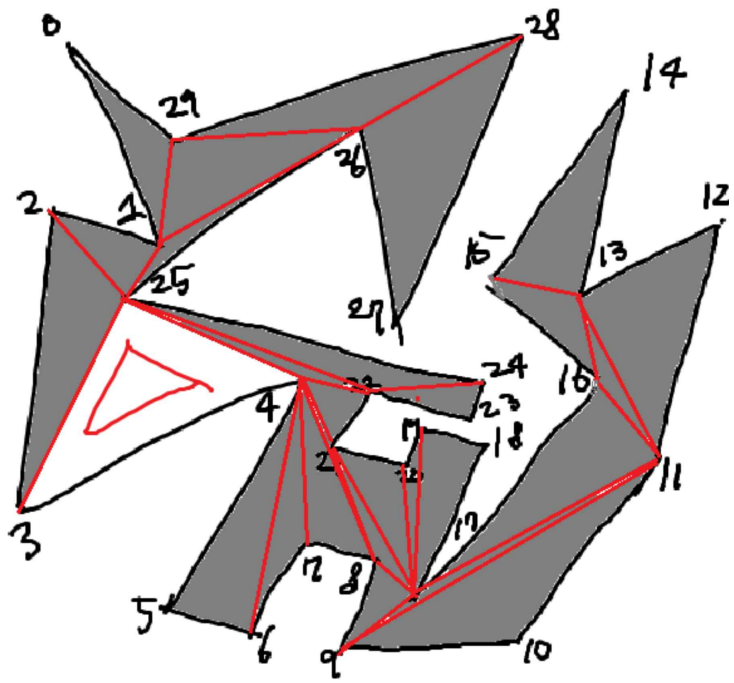
4번째 루프



5번째 루프



6번째 루프



이 과정에서 "다음 점"이라는 것은 삼각형이 하나 완성되면 바뀌게 된다. 때문에 수정/삭제가 자유롭고, 인근의 데이터에 쉽게 접근할 수 있어야 한다. 때문에 STL List을 사용해 해당 알고리즘을 구현했다.

문제 2 - 구멍이 뚫려 있는 경우.

하지만 모든 텍스트가 볼록하지 않은 다각형들의 모임으로 표현할 수는 없다.

구멍이 뚫린 경우가 있었기 때문이다.

"3D 게임 프로그래밍 1 - 2021180007 노훈철" 이라는 텍스트 안에서도,

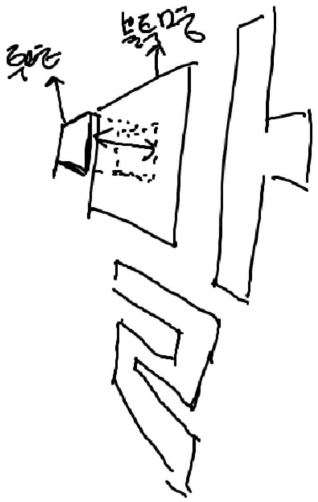
D, 임, 밍, 0, 8, 훈 의 경우가 그렇다. 이 경우에는 문제 1의 방법으로 해결할 수 없다.

때문에 다른 방법을 생각해야 했다. 이때 생각났던 방법은 "알파 블랜딩"의 오류를 역이용하는것이었다.

알파 블랜딩을 하기 위해서, 보통 불투명 물체들을 먼저 그리고, 그 후 반투명한 물체들을 카메라를 기준으로 정렬후, 뒤쪽부터 렌더링을 해야 했다. 그렇게 하지 않는다면,

앞의 투명한 물체가 뎁스 값을 업데이트 하기 때문에, 뒤쪽에 있는 불투명 물체가 그려지지 못한다. 이런 현상은 마치 구멍이 생긴것과 같다.

때문에 글자의 구멍들을 이렇게 처리할 계획이었다.



다만 아직 알파값을 처리하는 것이 잘 되지 않아서 그냥 검정색으로 처리를 했다.

이후 알파처리를 학습한다면 적용할 계획이다.

문제 3 - 삼각형들의 와인딩 오더 바꾸기

다각형을 일정한 순서대로 삼각형을 만들었음에도 삼각형들의 와인딩 오더는 다를 수 있었다. 애초에 점들의 와인딩 오더가 다르거나 하는 현상이 아닐까 싶다.

와인딩 오더를 수행하는 방법은, 삼각형의 법선을 세점으로 구성하고, 그 법선의 z 좌표

가 양수인지 음수인지 판별하고, 모든 삼각형에 대해 통일시킨다.

추가 구현내용 - 2. 인스턴싱 풀의 활용성을 높이다.

따라하기 예제에서 인스턴싱에 사용되었던 StructuredBuffer를 사용했었다.

단지 다양한 메쉬에 대해 인스턴싱 풀을 간편하게 추가할 수 있도록 구조를 약간 변형시켰다.

프로젝트에는 두가지 메쉬에 대해서 인스턴싱을 하는데,

폭발 파티클과 롤러코스터 레일에 대해서 작동한다.

또한 과제 1의 간단 오브젝트 풀을 이용해서 StructuredBuffer의 메모리를 관리한다.

때문에 할당하는 힙은 고정되어 있지만, 올리는 메모리의 양은 변할 수 있다.

```
struct InstancingStruct {
    ID3D12Resource* resource = nullptr;
    VS_VB_INSTANCE* InstanceDataArr = nullptr;
    CMesh* mesh = nullptr;
    void (*UpdateFunc)(InstancingStruct*, float) = nullptr;
    unsigned int Capacity = 0;
    unsigned int InstanceSize = 0;
    float extraData[6] = {};

    void Init(unsigned int capacity, CMesh* _mesh);
    void PushInstance(VS_VB_INSTANCE instance);
    void DestroyInstance(VS_VB_INSTANCE *instance);
};
```

위 그림은 Instancing을 위한 구조체의 모습이다.

Resource 는 힙, InstanceDataArr은 GPU에 올릴 메모리를 CPU에서 수정하기 위한 데이터다.

mesh는 인스턴싱을 할 메쉬

UpdateFunc은 인스턴스들을 업데이트를 할 함수이다.

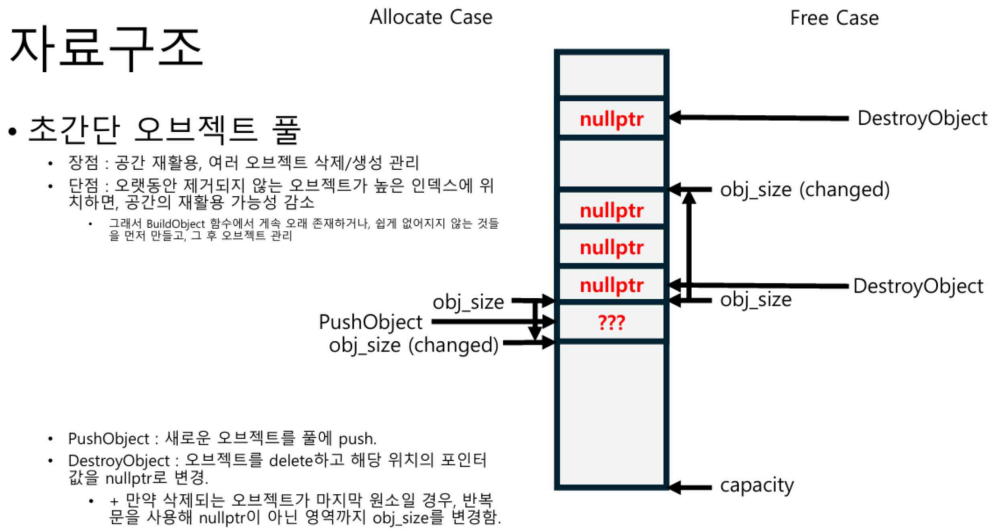
때문에 인스턴싱 되었더라도, 나름의 동작을 할 수 있다. 예를들어 폭발 파티클은 중심으로부터 점점 멀어지는 동작을 만들어낼 수 있었다.

extraData도 인스턴스들의 나름의 동작을 위해 쓰일 수 있는 인스턴스 전용 변수이다. 어떻게 쓰일지는 구현에 따라 다를 것이다.

InstanceSize는 GPU에게 넘길 데이터들의 개수를, Capacity는 힙안에 몇개의 인스턴스가 들어갈 수 있는지 표현한다.

Init 함수를 통해 용량과 메쉬만 넘기면 인스턴싱을 위한 구조체를 만들 수 있고,

PushInstance, DestoryInstance를 통해 오브젝트의 할당 / 해제를 하고, 비활성화된 데이터를 재활용 가능하다.



과제 1의 PushObject, DestoryObject와 그 구조가 같다.

인스턴싱에서는 nullptr 대신 ZeroMemeory를 사용했다. 그냥 메모리를 모조리 0으로 만들어서 월드 변환을 하더라도, 한점에 모이게 했다. (안보이게 된다.)