**게임 수학과 DirectX12 OT**

**한줄로 요약 : 3D 세상에 물체를 배치하고 그것이 우리의 2D 모니터 화면까지 출력되는 전 과정을 공부하는 학문**

유니티, 언리얼 엔진들이 이런 기능해주는데 왜 공부해야 하지?

→ 프로그래머는 항상 보이는 것에만 치중해서는 안되고 기반이 되는 근간 지식을 병행해서 공부를 해야 한다. 유니티로 게임 만드는 건 초등학생 한달 가르치면 가능해. 하지만 이건 상용엔진이 뛰어난 것이지 우리의 실력이 아니야. 그래픽스 공부하면 언리얼, 유니티 공부 안해도 당연히 게임 엔진에는 이런 기능들이 있겠구나 예측이 가능해. 게임 엔진을 바라보는 시각이 넓어진다.

회사 면접 보면 언리얼, 유니티 API 써봤냐는 질문 안해. 중요한건 기본기에 관한 것들이야.

선생님이 예전에 회사 면접에서 받은 질문들

- 렌더링 파이프라인?   
- Diffuse, Ambient, Specular?

- Double Buffering?

- Tangent Space?

- 그림자 원리?

- 짐벌락 현상은?

- Quaternion?

이와 같이 그래픽스 기본기에 대한 질문을 많이 받았어. 회사 입장에서는 10년 20년 써먹을 인재라 생각해서 뽑는건데 학생을 가르쳐서 따라잡힐 인재보다는 기본기에 충실하고 언리얼, 유니티, 혹은 자체 제작 엔진이건 어떤 포지션이든 환경과 조건을 가리지 않고 투입할 수 있는 그런 클라 개발자를 선언하는데 의외로 개발자 면접에서 이런 로우한 그래픽스 지식에 대한 질문을 많이 물어 본다. 면접 준비로도 훌륭한 강의

결국 물체를 화면에 그리는 과정까지 필요한 기초 수학과 렌더링 기법들을 배우게 될거야.

물체를 화면에 그려서 조명도 써보고, 그림자도 그려보고 파티클도 그려보면서 많은 실습을 하게 될거야.

코드도 단일 프로젝트에서 쌓아 올릴것이기 때문에 나중가면 방대한 코드를 작성하게 될 거야. 큰 규모에 대한 프로젝트에 대한 적응도 수월하게 하게 될거야.

한가지 당부사항: 유니티나 서버 강의부터 시작해서 새로운 API 함수들이 너무 많이 등장하다보니까 그런 부분에서 정신을 못차리는 분들이 많아. 자료구조나 알고리즘은 세밀하게 함수 하나하나를 따져보고 깊게 파고들어가는게 맞는데 게임 엔진이나 그래픽스나 운영체제 제작 같은 분야는 워낙 새로운 함수가 많이 등장하기 때문에 그렇게 함수 하나하나에 대해 새부적으로 깊게 이해할 필요는 없고 나무가 아니라 전체 숲을 보는 방식으로 공부를 해야 함. 그래서 선생님도 각 함수들에 대한 세세한 인자들에 대해서 일일이 설명하진 않을 것이고 관련해서는 마소 공식 홈페이지나 DirectX 12 관련 서적들을 참고 삼아서 같이 병행해서 읽는것을 추천 드려. 게임 엔진을 활용해서 만들 때 처럼 바로 결과물을 만들기는 힘들 수 있겠지만 장기적인 안목을 갖고 엔진을 지탱하는 기반 지식을 공부한다는 마음으로 천천히 시작을 해보자.

**그래픽스 원리**에 대한 오리엔테이션

DirectX를 이용해 뭘 해야 할지

비유로 이해하려고 노력을 해.

게임 클라이언트랑 영화 촬영하는 거랑 비슷하고

서버는 식당을 운영하는 거랑 느낌이 비슷해

유사한 부분들이 많아.

영화촬영 감독이 되었다고 가정해보자. 영화를 찍기 위해 필요한 건 장소부터 물색. 세트장 찾고. 조명 비추는 것부터 시작. 엔진 켜면 나오는게 영화 세트장이라 생각하자. 카메라, 라이트가 있어. 조명 끄면 어두워지고 카메라 끄면 아무것도 안보여. 카메라를 이용해 세상을 바라보는 상황.

1. 적당한 세트장

2. 조명

배우들을 알맞은 곳에 배치해야 해. 나무나 물품일 수 있어. 오젝트 큐브를 넣으면 소품 들어간거.

대본을 준비.

현실상에 있는 공간에다 이리저리 그럴사하게 배우들을 배치하고 카메라로 세상을 찍는게 영화촬영이라고 할 수 있어.

영화 촬영 현장에 가보면 카메라맨도 왔다갔다 하면서 찍는 그런 광경.

촬영된 결과물이 일반적으로 영화관에서 보는 영화.

영화 감독이 하는 것은 세트장, 배우 섭외하고 대본을 만들어서 어떤 특정 대사, 행동을 하게 만들어주고, 카메라맨한테 카메라를 들고 영화를 실시간으로 찍게끔, 부가적으로 조명도 신경써 줘야함.

이런 모든 조건들이 다 갖춰지게 되면 비로소 하나의 영화를 촬영할 수 있는 준비가 된거.

영화 촬영장 자체만 보면 3D 세상이야. 실질적으로 다양한 각도로 볼 수 있어. 하지만 실제로 영화가 찍힌 결과물. 카메라를 통해 찍힌 결과물을 보면 2D화면임을 알 수 있어. 즉 영화 카메라에서 카메라에 찍히는 장면 자체가 영화라는 걸 알 수 있어. 영화 보다가 고개 돌려도 다른 화면으로 전화되고 그런거 아냐. 영화 자체는 2D자체로 고정이 된 거라고 볼 수 있어.

2D 화면임에도 불구하고 3D화면을 보는 것 마냥 그럴사하게 보이는데 그 이유는 원근법이 제대로 잘 적용이 되고, 그림자들도 잘 적용이 되기 때문에 마치 3D 화면인것 같은 착시현상을 우리에게 주고 있어. 우리의 눈은 3D를 보는게 아니라 2D로 보고 있는 거. 카메라를 가까이 하면 커보이고 멀리하면 작아보이는 원근법 적용이 되고 물리적으로 일상적으로 경험할 수 있었던 모든 규칙들을 잘 지키면서 최종 결과물이 나오고 있어.

결국에는 게임이라는 것도 영화 촬영과 비슷해.

게임에 필요한 것은 무엇이냐.

배경에 조명 배치

예를 들어 Directional light, point light

게임 오브젝트 배치

카메라로 찍기 ->결과물이 게임 화면

게임을 플레이 한다는 것은 결국에는 실시간으로 카메라가 찍고 있는 그 모습 자체를 출력해주는 것이야. 게임 뿐만 아니라 애니메이션의 경우에도 동일한 기술이 적용된다고 할 수 있어.

영화, 3D애니, 게임 공통점은 결국 최종 결과물은 2D로 나온다는게 공통점

4K라는 걸 픽셀이 4000개가 대력적으로 있다는 이야기.

픽셀이란 화면을 엄청 확대해서 살펴보면 무수히 많은 점 단위로 이루어져있어.

4K란 가로로 4000개 세로로 대략 2000정도가 모여서 화면을 구성

영화건 게임이건 애니메이션이건 상관없이 이런 점들을 일일이 어떤 색상으로 그려줄지를 계산하는 게 최종 목표야.

영와, 애니, 게임 셋에는 어마어마한 차이가 있어.

영화를 촬영할 때는 단순하다고 볼 수 있는 건 물리적인 현상을 이용해 촬영을 하는 것. 카메라 하나만 있으면 모든 영상을 찍을 수 있어. 카메라로 물리적인 세상에 빛이 반사되어서 그 빛이 튕겨나와서 카메라에 상이 맺혀 지면서 상에 맺힌 이미지가 디지털 신호로 변경이 되어서 그것이 디지털카메라로 찍히게 될거야. 그래서 조명이든 어쩌구 저쩌구 배우들이 어쩌구 저쩌구 신경쓸 것 없이 그냥 결과물만 실시간으로 보면서 영화가 어떻게 찍혔는지 볼 수가 있어. 하지만 3D애니메이션, 게임의 경우 이야기가 달라지는게 애당초 세상 자체가 가상세계라는 것이 달라. 엄밀히 말하면 메모리 상에서만 존재하는 데이터들이야. 실생활에서는 그냥 카메라를 비춰서 결과물을 출력할 수 있는 것을 열심히 한땀한땀 수학 공식을 이용해서 현실속의 물리 법칙들을 연산을 통해 예측해서 최종 결과물이 이렇게 찍힐 것이다라는 것을 만들 수가 있어야 한다는 이야기가 되는거고. 결국 그게 게임 혹은 3D 애니메이션이라고 할 수 있어.

영화와 비슷하지만 다른건 찍는 과정은 비슷하지만 결과물이 도출되는게 물리적인 법칙을 이용하는 거냐 수학적으로 한땀한땀 계산해야 하느냐의 차이야. 그게 사실 어마어마하게 큰 차이야.

결국 애니메이션과 게임은 영화보다 복잡한 과정을 거쳐야 하는데 설상가상으로 게임은 애니메이션보다 극악의 난이도야. 애니메이션은 한번만 찍으면 됨. 1만대의 컴퓨터로 한번만 딱 잘해주면 장땡인데 그 이유는 애니메이션을 볼 때 고개 돌려도 변하지 않기에 한번 계산하면 영구적으로 유지되는 경우인데 게임의 경우는 묘하게 카메라가 고정이 아니야. 실시간 렌더링이라고 해. 게임에서는 카메라 위치가 계속 변해. 롤의 경우는 탑뷰로 위에서 아래를 가리키고 오버워치의 경우는 플레이어의 이마쪽에 카메라가 있어서 플레이어의 위치에 따라 카메라가 계속 이동하게 될거야. 그러다 보니 플레이어가 조종하는 것에 따라 카메라의 위치가 바뀌다보니 변화하는 카메라의 좌표와 각도에 따라 계속 연산을 다시 해줘야 하는 어마어마한 일을 해줘야 해.

영화, 애니, 게임의 차이는 결국에는 실시간으로 하냐 아니냐의 차이도 있다고 볼 수 있어.

게임같은 경우 그래픽스의 기술을 아주 최신까지는 활용을 할 수 없는게 실시간으로 매번마다 갱신을 해야하다보니 하드웨어적으로 한계가 있어서 일반적으로 최신 그래픽스 기술에 대해서 5~10년 정도 뒤떨어진 기술을 사용할 수 밖에 없어. 실력이 딸려서가 아니라 하드웨어가 실시간으로 계산하는 것을 버티지 못하기 때문이야.

결론은 DirectX이용해서 무엇을 해야 하느냐 세트 촬영장에 가상의 물체들의 정보를 이용해서 화면을 최종적으로 계산할수 있어야 해.

계산하라고 하면 못해. 어마어마하게 많은 연산량이 필요하게 됨. 그러다 보니 게임 수학과 DirectX를 배우게 되면 일련의 과정들이 어떻게 처리 되는지 이해하게 될 것이고, 3D화면이 어떻게 2D화면으로 보여지는지 연구하는 학문이 그래픽스라고 이해하면 됨.

굉장히 오랜기간 연구된 학문이기에 어느정도 체계가 잡혀있어. 그걸 한땀한땀 어떤식으로 접근해야지 3D화면이 2D로 변환되는지 알아볼거야.

**렌더링 파이프라인 OT**

영화 감독인데 디지털 카메라가 고장났다고 가정. 엄청난 수학천재여서 수학계산으로 모든 세상을 그려본다 가정을 해보자. 여기의 모든 빛을 점단위로 계산하는 건 말이 안됨. 점이 너무 많기 때문에. 카메라가 바라보는 이 세상의 범위의 물체들을 대상으로 연산을 해야 하고. 어떤 물체를 옮겨야 할 때 어떤 정보가 필요할까? 모든 픽셀 단위로 점의 색깔을 맞출 수 있으면 그림이 나오는 것이긴 한데 그 그림을, 색상을 그리기 위해서는 어떤 물체가 어디로 이동을 하는지를 알아야 해. 상자가 이동할 때 마다 좌표가 변하는걸 알 수 있어.상자라는 것 자체가 지금은 이어진 물체처럼 보이는데 엄밀히 말해서 게임 세상에서는 모든 물체들이 다 대부분 삼각형으로 이루어져있어. 꼭 삼각형이어야 하는건 아니지만 통상적으로 삼각형으로 만들어. Shading Mode를 Wireframe으로 바꾸면 삼각형 단위로 물체를 보여주고 있는데 큐브도 일련의 삼각형의 연속으로 이루어져있어. 삼각형의 정점은 8개가 있을거야. 8개의 정점에 대한 좌표 연산을 통해서 각 정점이 어떤 위치에 올지를 계산할 수 있어야 해. 현실상에선 복잡한데 포개져서 앞에 있는 큐브에 의해 뒤에 큐브가 안보이는 상황 발생할 수 있어. 정확히 말하면 위치 계산을 하면서 누가 더 앞에 오느냐도 계산을 해줘야 함. 결국에는 색상도 중요하긴 한데 색상을 알기 위해서는 이 물체의 좌표를 위치를 변환할 수 있어야 해. 큰 틀에서 보면 위치와 색상과 관련된 계산을 해줘야 한다는 거고. 모든 물체들은 삼각형 단위라고 했었어. 캐릭터는 삼각형 아닌거 같아. 근데 가까이서 보면 삼각형으로 이루어져있는걸 알 수 있어. 큐브를 옮기기 위해 정점이 8개 있었는데 캐릭터는 몇천개의 정점으로 이루어져 있다는 것을 알 수 있어. 연산하기 귀찮을 거야.

카메라 범위 있는 물체들을 2D로 옮기는 공식을 찾았다고 가정. 근데 거기서 문제가 끝나는게 아니야. 공식을 알아도 실질적으로 하는건 별개야. 삼각형들을 엄청나게 늘어날 수 있어. 수풀, 나무, 캐릭터, 몬스터, 미사일도 날아간다면 모든 삼각형들을 일일이 연산을 통해 좌표 계산을 해주는게 힘들거야. 영화 감독이라 가정했을 때 할일이 많아. 화면 좌표를 2D 화면에 옮기는 것도 중요한 작업이긴 하지만 이것 말고도 전반적인 게임에 대한 규칙을 관리해야 함. 상자는 앞으로 조금 이동해주세요. 캐릭터는 고개를 틀어 딴데를 봐주세요 다 계산해야 하고 캐릭터가 공격해서 체력 깍이면 언젠가 소멸하는거 다 해줘야 해. 게임을 굴리기에도 할일이 많아. 감독이 할 일이 너무 많은데 한땀한땀 노가다까지 하는건 너무 귀찮은 일이 될거야. 자본주의 사회에서는 외주주면 모든게 해결이 됨. 단순하게 계산을 빨리 잘하는 계산 모든 것을 떠넘기면 됨. 결국 영화 감독에 해당하는 게임 규칙 정하는 연산을 하는 핵심적인 부분을 관리하는 건 CPU의 역할이고 그래픽과 관련된 노가다 작업을 당담해서 3D 세상에서 2D 세상으로 넘어가기 위한 변환 연산을 해주는게 GPU의 역할이라고 볼 수 있는 거 이렇게 역할 분담을 확실히 한다고 볼 수 있어. GPU를 꼭 사용해야 하는 건 아냐. 고전게임 같이 정점 개수가 몇 개 없는 캐릭터들이라면 CPU가 계산을 해도 문제가 없는데 CPU는 자체로도 할일이 많다보니까 현대적 게임에서는 CPU가 렌더링까지 다 계산을 한다는건 수지타산이 맞지 않는 거. 그래서 그런건 GPU에 다 떠넘긴다는 것이 첫번째 결론

**GPU와 렌더링 파이프라인**

이게 나오는 이유는 CPU가 모든 것을 떠맞아서 하기 보다는 적절하게 외주를 떠넘기는 작업을 하기 때문이야. 외주 스토리가 이어지는 것. 게임 세상에 있는 모든 물체들에 대한 연산 자체를 CPU가 하기엔 너무 바쁘다보니 GPU와 CPU 부품끼리 각각 역할을 분담한다고 볼 수 있어.

가끔 가서 면접에서 나오는 질문인데 CPU와 GPU의 차이에 대해 이해하고 있어야 해.

CPU같은 경우는 고급 인력. 복잡한 연산을 굉장히 잘하고 굉장히 뛰어난 기억력을 가지고 있어. 만능형 인제. ALU라고 되어 있는 부분이 인간으로 치면 실질적으로 산술연산을 하는 두뇌에 해당하는 영역인건데 ALU라는 영역 자체가 어마어마 하게 많지는 않다는 것을 알 수 있어. 즉 CPU 자체는 개개인으로 보면 실력이 뛰어난 소수 집단이라고 볼 수 있는 거고. GPU는 반대로 대부분의 영역 자체를 ALU가 차지하고 있음을 알 수 있어. 계산기 같은 역할이야. 계산기는 사람보다 똑똑하다고 할 수 있을지 모르겠지만 단순 계산을 빠르게 잘할 수 있는건 확실해. 값싼 인력들이 모인 인력 사무소 같은 느낌의 장치라고 볼 수 있어. 복잡하지 않은 연산을 아주 많은 수의 ALU들이 붙어서 떼거지로 하는 그런 이미지를 기억해주면 돼.

CPU는 범용적으로 연산을 하게 되고, 운영체계를 실행하고 게임을 키고 문서 작업을 할 때 등 범용적인 부분에 CPU가 개입을 하게 되고, GPU는 연산을 떠넘기게 되는데 서로 연관성이 없는 독립적인 일감을 떠넘길 때 굉장히 유용해. 연관성이 없는 일들을 병렬로 처리한다는데 왜 연관성이 없는게 중요한지 언급을 해보자면 일단 대표적으로 일감들이 암호학이나 인공지능이나 비트코인체굴 등 이런 연산에 유용해. 비트코인 체굴이나 암호학 같은게 수학적 난제를 푸는 것과 비슷해. 어떤 암호를 대입해 그 암호가 맞는지 체크하는 그 연산 자체는 100, 200, 300으로 테스트하다 그 세가지 연산을 다 독립적으로 실행할 수가 있어. 마찬가지로 인공지능 같은 경우도 바둑 인공지능을 만든다면 돌을 놓는 수가 많은데 어떤 지점을 기준으로 왼쪽에 돌을 놓느냐 오른쪽에 놓느냐 두가지 갈림길이 있다고 했을 때 두 갈림길은 각각 서로 독립적인 연산이야. 왼쪽에 놓는 연산이 오른쪽에 영향을 주지는 않아. 그러다보니 독립적인 역할을 일감을 분배해서 여러 코어들이 협동해서 하나하나씩 당담을 해서 계산을 할 때 이 GPU가 빛을 본다고 보면 됨. 반면 연산 중에서 어떤 연산을 한 다음에 그 연산 결과물에다가 또 다른 연산을 하고 이렇게 연산이 독립적이지 않고 계속 이어져야 한다면 GPU는 힘을 못쓰는 부분이 있다. CPU가 일반적인 상황에서 유용하고 GPU는 정말로 서로 연관성이 없는 하지만 일감 자체는 많은 일을 할 때 빛을 본다고 보면 됨.간단한 계산기라고 생각하면 됨. 외주에 적합하다고 볼 수 있어.

입장을 바꿔서 게임으로 돌아가면 게임에서 연산하는 걸 GPU에 떠넘기면 왜 좋은지 생각을 해보자. 독립적인 일감을 병렬로 실행할 때 유용하다고 했는데. 사실 이 게임에 있는 여러 물체들이 각각 독립적으로 연산을 할 수 있어. 물론 여기에 조명이 들어간다거나 하면 얘기가 달라지지만 각각의 물체들은 서로에게 영향을 당장은 주지 않는다. 그렇기 때문에 첫번째 큐브가 변환이 되면 어떤 좌표에 들어갈지 계산하는거나, 두번째, 세번째 큐브 나머지 플레이어의 좌표 계산을 독립적으로 할 수 있다보니 이 GPU에 모든 일감을 떠넘기면 펼리하다. 언젠가는 결과를 합치겠지만 CPU가 한땀한땀 하기 보다는 GPU가 병렬로 쫙 한번에 하는게 효율적이야. 픽셀 단위로 어떤 색을 입힐지도 수학적 공식을 이용해 계산 해야 하는데 CPU가 각 픽셀 돌면서 하는 건 말이 안되는 거고 그것고 GPU가 병렬로 쫙 실행해서 결과물만 합체면 빠르게 동작해. 그러다 보니까 GPU가 암호학이나 비트코인 체굴, 인공지능에도 유용하지만 근본적으로 처음에 개발하게 된 계기는 그래픽스와 관련이 있어. 컴퓨터 그래픽에서 GPU가 있으면 일감을 떠넘기기 좋은 외주 시스템이다 보니까 발전해왔다고 볼 수 있어. 여기서 중요한 것은 일감을 GPU에 떠넘기는 것 까지는 좋은데 우리 멋대로 대충 이렇게 해봐라고 명령을 내릴 수는 없어. GPU도 정해진 스펙에 맞게 정해진 연산을 하는 것을 좋아하기 때문에 아무렇게나 대충 외주를 날길수는 없고 정해진 규칙에 따라 외주를 날려야 됨. 결과적으로 3D화면 자체를 메모리 정보를 토대로 2D화면으로 결과물을 출력하기 위해 일련의 단계들을 전문가들이 연구를 해왔고 그 단계가 바로 랜더링 파이프라인이라고 불리는 단계야. 지금 다 외울 필요는 없고 조금씩 영역을 확대해가며 살펴보게 될 거아. 다 사용할 것은 아니고 Vertex Shader 단계를 제외하고 몇가지 애들은 옵션으로 해도되고 안해도 되는 그런 부분이기 때문에 당장 사용해야 하는 건 아니야. 간략하게 언급을 해보자면

1. 처음에 **Input** **Assembler** 단계가 있는데 **정점 정보**를 우리가 **전달**을 하는 단계라고 볼 수 있어. 어떤 정점들이 어떻게 연결이 되어 있는지. 예를들어 큐브의 삼각형이 어떻게 이루어져 있는지를 Input Assembler 단계에서 전달을 해준다.

Assembler: 명령을 기계어로 전환하는 프로그램, 조립공

2. **Vertex** **Shader** 단계가 두번째인데 소위 VS라는 약자로 기억을 해주면 됨. Input Assembler 단계에서 넘겨준 정점들을 대상으로 연산을 하는 부분이 여기서 들어가 됨. 여기서 정점을 대상으로 연산에 들어가게 되는 거. 예를 들어 해당 좌표의 정점을 이동해야 한다하면 변환이 일어나야 한다. Spinnig, morphing 이런 고급 기법들이 들어갈 때 그런 부분들을 두번째 단계 Vertex Shader에서 해준다고 보면 된. **정점 변환**을 해주는 거다라고 한준 요약을 해줄 수 있어. 카메라가 어떠한 특정 각도로 바라볼 때 대략적으로 어떤 위치로 변하게 된다라고 공식화 해줘서 그걸 해주고 있다고 보면 됨.

3. 3총사가 나오는데 뭉뚱그려서 **Tessellation** 단계라고 함. (모자이크 세공)

**Hull Shader**

**Tessellator**

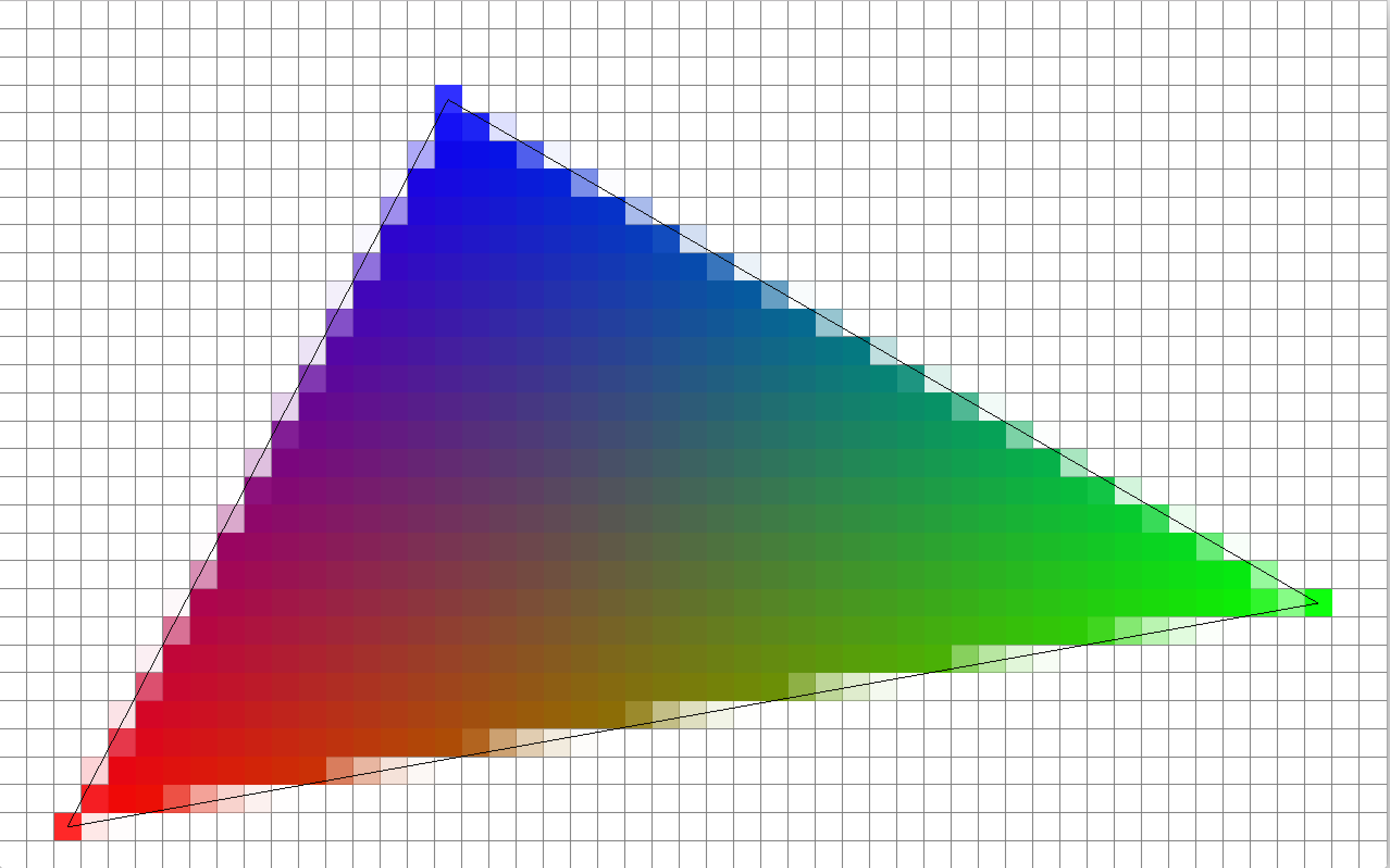
**Domain Shader**

이 세개가 있는데 얘네들은 DirectX11에서 추가된 개념.

Tessellation과 Geometry는 공통적으로 새로운 정점들을 생성할 때 활용한다는 특징이 있어. 정점을 왜 추가하는 것이냐. 외주를 맞겼는데 경우에 따라 조금 보정을 하고 싶다고 요청을 해본 상황이 있다고 가정을 해보면 곧이 곧대로 뭔가를 그려준 정점 뿐만 아니라 원하는 정점을 추가해서 정점 개수를 늘리는 방법으로 뭔가 하는거. 어떨 때 이용하냐. MMORPG 게임에서 어떤 지형이 있다고 가정. 지형도 일련의 삼각함수로 이루어져 있는데 저 멀리 있는 삼각형가지 새새히 보일 필요 없어. 가까이 있는 지형의 경우 조금 더 선명하게 보이는게 아무래도 유저들이 받아들일 때 뛰어난 그래픽으로 받아들일 확률이 높아. 멀리 있는 애들은 흐릿하게 보이고 가까이 있는 애들을 정점들을 추가해서 더 면밀하게 보이게 할 때 이런 Tesselation 구조를 활용할 수 있어. Tesselation, Geometry 둘다 정점을 추가하는 개념이긴 한데 Tesselation은 좀 더 거시적인 관점. 지형 등등에 이용하고 Geometry Shader는 조금 더 작은 단위 삼각형 하나를 대상으로 작업하거나 할 때 이용한다고 보면 됨.Direct9에서는 존재하지도 않았던 애들이야. 그러다 보니 DirectX9에서는 Vertex Shader에서 바로 건너 뛰어서 Rasterizer로 넘어 왔었어.

4. Geometry Shader 이 애는 DirectX10에서 추가된 애

5. Rasterizer는 중요하니 자세히 언급. 이단계 전까지는 정점을 대상으로 계속 연산을 했던 거야. 정점을 대상으로 연산을 해서 이 삼각형이 어떤 위치에 어떤 색상으로 만들어야 하는지 알 수 있었어. 근데 세부적으로 모든 점들을 대상으로 연산을 한게 아니라 각 꼭지점 정점 대상으로 3개만 좌표랑 색상이 확정 되었다고 하면 삼각형이라는 걸 아니까 삼각형을 대략적으로 만들어서 보관하는 작업이 필요해. 각 꼭지점이 빨, 파, 초라고 하면 색상을 적절히 섞어서 그레디에이션을 준 거 볼 수 있어. 마찬가지로 픽셀 단위로 모든 작업이 일어나다 보니까 대각선으로 긋는 작업도 결국에는 점 단위로 표현을 하다 보니 적절하게 픽셀의 색상을 구별한 것을 볼 수 있어. 이렇게 해야지 부드러운 삼각형을 볼 수 있어. 이렇게 하는 단계를 Rasterizer 단계라 볼 수 있어. 정점 데이터를 기반으로 픽셀로 변환하는 작업들. 내부 점들을 보관하는 그런 작업이 일어난다고 볼 수 있어.



6. Pixel Shader 는 최종적으로 색상을 입히는 단계라고 볼 수 있어. 색상을 조절하거나 강제로 빨간색으로 만들어 준다거나 색상 변경을 Pixel Shader에서 해줄 수가 있어.

10. 최종 결과물을 받았으면 Output Merger 라는 단계에서 마무리 단계로 모든 정보를 조합해서 최종 색상을 결정해주는 일련의 과정들로 이루어져 있어.

처음에 이 과정들을 다 이해할 필요 없이 굉장히 많은 정해진 규격에 따른 그런 단계들이 있다는 것만 이해를 해주면 됨.

마지막으로 CPU와 GPU 사이에 왔다갔다 핑퐁을 하면서 화면을 그려주는 요청을 계속 해주면서 결과물을 받아서 그려주고 다음 연산을 하는 식으로 굴러가는 걸 어렴풋 알겠는데 GPU도 제조사마다 다 사양이 달라 GPU한테 외주를 맞겨야 하는 입장에서 GPU마다 모든 스펙이 다르면 당장 작업을 할 때 어려울거야. 어떤 회사에서 만든 GPU에서는 이런 요청을 하고 버그가 발생하고 사양이 바뀌면 외주하는 작업 자체에 집중할 수 없어. DirectX라는건 마이크로 소프트에서 제공하는 GPU외주 라이브러리 라고 볼 수 있어. GPU에 일감을 떠넘길 때 DirectX라는 라이브러리를 통해 가지고 요청을 하게 될 거고 GPU제조사들을 마이크로 소프트사와 협력을 해서 DirectX 스펙이 맞춰서 GPU를 제조를 한다고 보면 됨. 우리는 GPU를 어떤 회사에서 만들었건 전혀 생각할 거 없이 그냥 DirectX 스펙에 맞게 코드를 작업을 하면 나머지 부분은 MS사와 GPU제조사에서 열심히 맞춰야 하는 부분이 된다고 보면 됨.

결국에는 게임 화면을 메모리에서만 들고 있는데 이것을 이쁘게 2D 화면으로 출력하는 작업을 이제 강의에서 이어서 쭉 이어서 하게 될거야.

