Alkanoid 3D

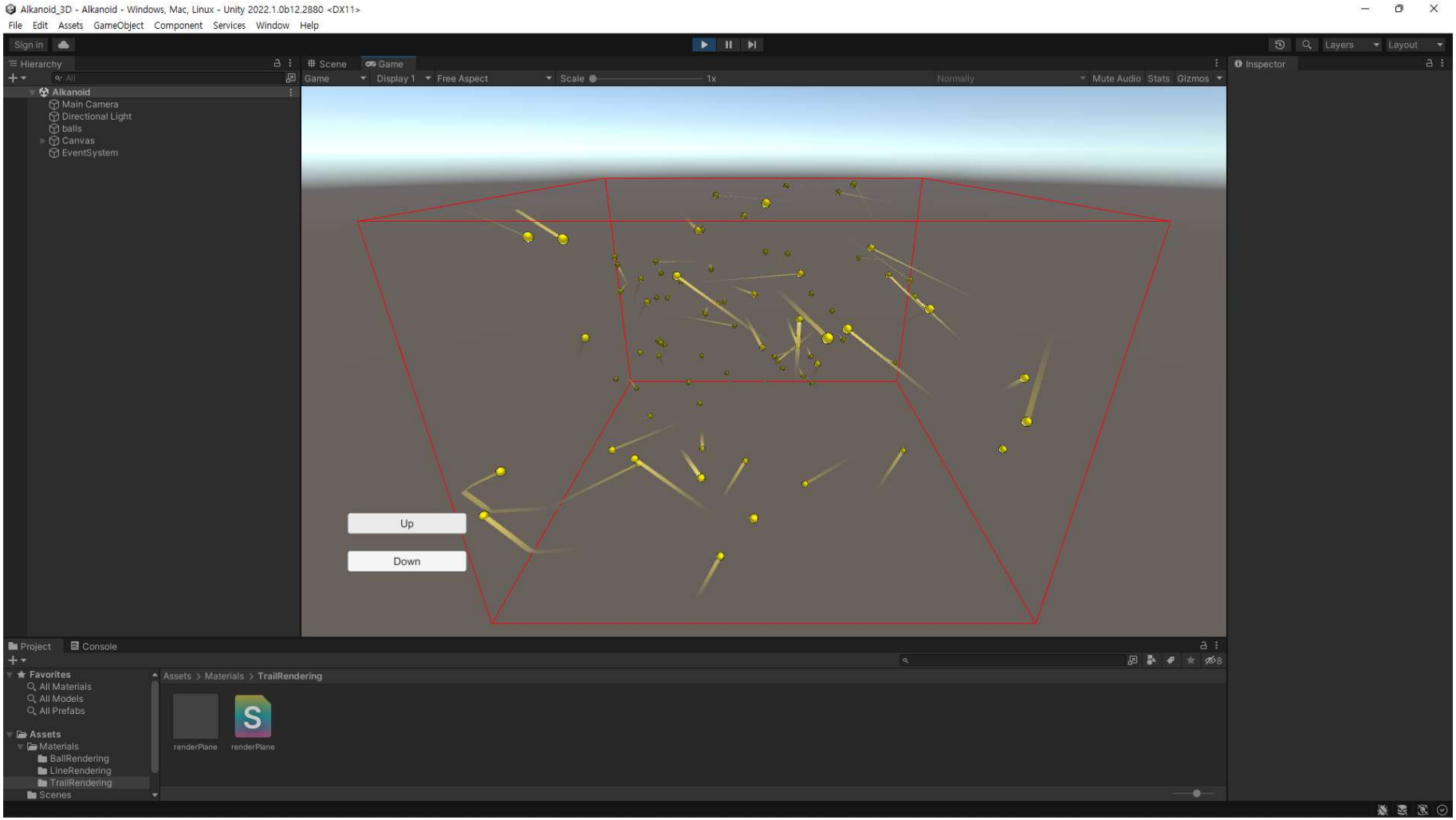
tspaghettit@gmail.com 김동현

목차

1. 개요

- 갇힌 3차원 공간 내에서만 움직이는 공들을 여러 개 만드는 프로젝트
- 버튼을 통해서 공 1개 당 2개씩 생성할 수 있다
- 반대로 2개씩 사라지게도 할 수 있다
- 공이 날아가는 궤적을 따라, 흔적을 나타내는 Mesh 를 생성해 공의 경로를 따라간다

구현 내용 (https://youtu.be/-stKSOqev48)



2. 이 프로젝트를 통해 얻고자 한 것

- Unity 엔진에서 Compute Shader 작성 및 사용에 대한 연습
- 그래픽스 파이프라인에 대한 공부
- Procedural Mesh 생성 경험
- 날아가는 듯한 효과를 주는 그래픽을 만들어 보는 경험

3. 구현 내용과 구현 방법

A. Ball 객체

- Compute Shader 를 이용해서, 각 Ball 의 움직임을 계산 및 제어
  - 하나의 Thread 가 하나의 Ball 연산을 담당
- 최대 그릴 수 있는 공의 개수만큼 한번에 버퍼를 할당
- 버튼을 누를 때마다 렌더링 되는 공의 개수가 증가
  - 기존 공 1 개당 2 개씩 새로운 공이 갈라져서 나옴
  - 좌, 우로 1개씩 총 2개의 공이 새롭게 생성된다

A-1. C# 스크립트

```
/*
Ball 하나를 나타내는 구조체
객체 하나 = 3 + 3 + 1 + 3 + 3 + 3 = 16 * float
*/
참조 3개
public struct Ball
{
    //position, velocity = 월드 공간 기준
    public Vector3 position;
    public Vector3 velocity;
    public float speed;

    // 회전 후 모델의 기본 좌표계
    public Vector3 xBasis;
    public Vector3 yBasis;
    public Vector3 zBasis;
}
```

1. Ball 데이터를 저장하는 구조체
   - Ball 의 위치, 속도 데이터를 저장
   - Ball 객체의 모델 좌표계 벡터를 저장
     - Ball 의 속도 방향에 맞춰서 Ball 모델 좌표계를 회전시킨다

```
void increaseDrawNumber() // click number 가 증가했을 때, 그릴 공의 정보를 늘리는 것
{
    /*
    한번 클릭할 때마다, 1 개의 공마다 2개의 공이 새로 생성된다
    Binary Tree 구조하고 비슷, 부모 공 하나마다 2개의 자식 공이 생성

    [startIndex, endIndex) = 새로 생성되는 공의 인덱스
    */
    //새로 생성되는 공의 인덱스 계산
    startIndex = endIndex;
    endIndex *= 3;

    //공의 개수가 버퍼 사이즈를 넘어가는 경우, 2 가지 다 고려를 해야 한다
    //먼저 endIndex 가 버퍼 사이즈 자체를 넘어가는 경우
    //endIndex = (endIndex >= ballBufferSize) ? ballBufferSize : endIndex;


    //버퍼 사이즈 고려할 필요 없이, 공의 개수가 최대 개수를 넘으면 그냥 무조건 최대 개수만 가지도록 한다
    endIndex = (endIndex >= ballsCountMax) ? ballsCountMax : endIndex;


    //새로운 정보를 구하고자 하는 공의 범위를 구했으니, compute shader 에 값을 갱신한다
    computeShader.SetInt("startIndex", startIndex);
    computeShader.SetInt("endIndex", endIndex);

    //세로 추가된 Ball 들의 정보를 구하는 커널 함수 실행
    computeShader.Dispatch(ballKernelHandleAssign, groupSizeX, 1, 1);

    // 새로 구한 draw number 를 argument 배열에 넣어주고, arg 도 갱신해준다
    args[1] = (uint)endIndex;   // GPU Instancing 을 통해 그릴 Instance 개수 갱신
    argsBuffer.SetData(args);
}
```

2. 그리는 Ball 의 개수를 늘리려는 경우

- Ball 의 데이터를 저장한 버퍼 내에서, 추가로 그리고자 하는 범위의 인덱스를 지정
- 기존의 1 Ball 하나당 2 개의 자식이 생긴다

```
void decreaseDrawNumber() // click number 가 감소했을 때 처리
{
    //이번에는 반대로 줄이고자 하는 범위를 구한다
    //최소 0~1 범위까지만 내려가야 한다
    startIndex /= 3;
    endIndex = (endIndex / 3 == 0) ? 1 : endIndex / 3;

    // 줄어든 범위를 compute shader 에 갱신한다
    computeShader.SetInt("startIndex", startIndex);
    computeShader.SetInt("endIndex", endIndex);

    //공의 개수가 줄어드는 경우에는, 호출할 커널 함수가 없다.
    //그냥 그리는 Instance 개수를 줄이면 된다
    //새로 구한 draw number 를 argument 배열에 넣어주고, arg 도 갱신해준다
    args[1] = (uint)endIndex;
    argsBuffer.SetData(args);
}
```

3. 반대로 Ball 의 개수를 줄이는 경우

- 기존 범위에서 3배가 된 것이 추가된 범위 였으므로, 이번엔 반대로 범위를 3배로 줄인다
- 그리고자 하는 Instance 개수도 갱신

A-2. Compute Shader

```
//공의 움직임을 계산하는 커널 함수
//공의 위치, 속도 벡터 모두 월드 공간 좌표계 기준 값이라 가정하고 진행한다
[numthreads(GROUP_SIZE, 1, 1)]
void moveBalls (uint3 id : SV_DISPATCHTHREADID)
{
    //담당 스레드가 그리려는 공보다 높은 숫자이면, 계산하지 않는다
    if(id.x >= (uint)endIndex) return;

    //계산하고자 하는 공에 대해서
    Ball ball = ballBuffer[id.x];

    //현재 공이 움직일 수 있는 영역내에 존재하는 지를 먼저 확인한다
    if(abs(ball.position.x) <= abs(limitSize.x) &&
       abs(ball.position.y) <= abs(limitSize.y) &&
       abs(ball.position.z) <= abs(limitSize.z)) {
        //영역 내에 존재한다면, 따로 처리할 내용은 없다
    }
```

1. Ball 의 움직임을 계산하는 커널 함수
   - 먼저 해당 Thread 에 할당된 Ball 이 그려지는 공의 범위에 속하는 지 파악
   - 해당 Ball 의 위치가 움직일 수 있는 영역에 존재하는 지 확인

```
else {
    //공의 위치, 속도 벡터 모두 월드 공간 기준
    /*
    공이 정해진 영역을 벗어났더라도
    정해진 영역으로 다시 들어오는 중인지, 계속 벗어나는 중인지를 판단해야 한다

    따라서 공이 영역을 벗어났지만, 여전히 밖으로 나가고 있는 영역을 파악해서
    해당 방향을 향하는 노멀 벡터를 만든다

    이후 공의 속도 방향을 노멀 벡터 기준으로 반사 시켜~
    영역으로 들어오는 방향을 가지게 한다
    */

    //정해진 영역으로 들어오기 위한 반사 노멀벡터
    float3 reflectNorm = float3(0,0,0);

    //영역을 벗어나는 경우들을 파악해서 반사 노멀벡터 값을 구한다
    if(ball.position.x > limitSize.x) reflectNorm.x += (ball.velocity.x < 0) ? 0 : 1;
    if(ball.position.x < -limitSize.x) reflectNorm.x += (ball.velocity.x > 0) ? 0 : 1;

    if(ball.position.y > limitSize.y) reflectNorm.y += (ball.velocity.y < 0) ? 0 : 1;
    if(ball.position.y < -limitSize.y) reflectNorm.y += (ball.velocity.y > 0) ? 0 : 1;

    if(ball.position.z > limitSize.z) reflectNorm.z += (ball.velocity.z < 0) ? 0 : 1;
    if(ball.position.z < -limitSize.z) reflectNorm.z += (ball.velocity.z > 0) ? 0 : 1;

    //반사 노멀벡터가 0 벡터가 아닌 경우 = 속도를 바꿔야 하는 경우
    if(length(reflectNorm) != 0) {
        //정규화
        normalize(reflectNorm);

        //노멀 벡터에 대한 반사를 통해, 돌아오는 방향으로 속도를 갱신한다
        ball.velocity = reflect(ball.velocity, reflectNorm);

        // 바뀐 속도에 대해서 모델 좌표계를 다시 만든다
        float3 UP = float3(0,1,0);
        ball.zBasis = normalize(ball.velocity);
        ball.xBasis = normalize( cross(UP, ball.zBasis) );
        ball.yBasis = cross(ball.zBasis, ball.xBasis);
    }

}
```

- Ball 이 정해진 영역을 벗어났다면, 공의 속도 방향을 벗어나는 영역에 대하여 반사 시켜야 한다
- 벗어나는 범위를 파악해, 노멀 벡터를 구하고
- 해당 노멀 벡터를 이용해서 속도 반사
- 바뀐 속도를 이용하여, 모델 좌표계 축 벡터를 갱신한다

```
// 최종 속도에 대해서 이동 실시
ball.position += ball.velocity * ball.speed * deltaTime;

// 갱신한 공의 정보를 버퍼에 넣어준다
ballBuffer[id.x] = ball;
```

```
//[startIndex, endIndex) 범위 인덱스 Ball 에 데이터를 할당하는 커널 함수
[numthreads(GROUP_SIZE, 1, 1)]
//SV_DISPATCHTHREADID => 전체 그룹 내에서 해당 스레드의 id
void assignBalls (uint3 id : SV_DISPATCHTHREADID)
{
    /*
    스레드 하나 = 버퍼의 Ball 하나 담당
    스레드 id 와 버퍼의 인덱스는 같은 Ball 을 담당한다
    */
    //해당 스레드가 새로 생겨서, 값을 할당받을 공이 아니라면, 처리할 작업이 없다 => 종료
    if(id.x < (uint)startIndex || id.x >= (uint)endIndex) return;

    //해당 스레드에 랜덤한 값을 하나 가져온다 => [0, 1] 범위 값
    float rndVal = random(id.x, deltaTime);

    /*
        먼저 Ball Buffer 부터 채운다

        해당 공의 부모 공 인덱스 = (id - startIndex)/2
    */
    //위치 = 부모 공에서 갈라져서 나오기 때문에, 같은 위치를 가진다
    ballBuffer[id.x].position = ballBuffer[ (id.x-startIndex) / 2 ].position;
```

2. 추가된 Ball 들에 데이터를 할당하는 커널 함수

- 새로 추가하고자 하는 범위에 해당하는 Thread 인지 확인 한다
- 새로 생기는 Ball 2 개는 기존 1 개의 Ball 에서 갈라져 나와야 하기 때문에, 부모 인덱스의 위치와 같은 값을 가져야 한다

```
//velocity = 현재 공의 속도 방향에서 살짝 왼쪽 or 오른쪽으로 비튼 방향을 준다
//2개의 자식이 서로 다른 방향으로 속도를 가지도록 한다
float flag = ((id.x-startIndex) % 2) == 0 ? -1 : 1;

//부모 공의 속도 값을 기본적으로 가진다
//앞서 구한 랜덤 값을 이용해서 자식 공의 방향을 조금 수정한다
ballBuffer[id.x].velocity = ballBuffer[ (id.x-startIndex) / 2 ].velocity
                //부모 공의 모델 좌표계 값을 이용해서, 자식 공의 새로운 방향을 설정
                //x축 방향 = 나아가는 방향 기준 좌,우 를 의미 => flag 값을 이용해서 좌, 우 를 설정한다
                + flag * ballBuffer[ (id.x-startIndex) / 2 ].xBasis * rndVal
                        //랜덤 값이 [0,1] 범위 값이기에 y,z 축 방향으로 [-1, 1] 범위의 랜덤한 값을 곱해
                        //조금 다른 방향을 가지게 한다
                        + ballBuffer[ (id.x-startIndex) / 2 ].yBasis * (rndVal * 2 - 1)
                        + ballBuffer[ (id.x-startIndex) / 2 ].zBasis * (rndVal * 2 - 1);

//방향이 0 이 되는 상황은 피한다
if( length(ballBuffer[id.x].velocity) == 0 ) ballBuffer[id.x].velocity = float3(1,1,1);

//속도는 항상 정규화 시켜서 저장한다
ballBuffer[id.x].velocity = normalize( ballBuffer[id.x].velocity );

//속력은 각자 랜덤하게 설정한다, 아예 움직이지 않는 상황은 피한다
ballBuffer[id.x].speed = (rndVal + 0.01) * maxSpeed;
```

```
//자식 공의 모델 좌표계 설정
//공의 월드 공간 좌표계 기준으로 설정 => 업데이트한 속도를 바탕으로 설정한다
float3 UP = float3(0,1,0); //월드 공간 기준 y축
ballBuffer[id.x].zBasis = normalize(ballBuffer[id.x].velocity); //z = 나아가는 방향
ballBuffer[id.x].xBasis = normalize(cross(UP, ballBuffer[id.x].zBasis));
ballBuffer[id.x].yBasis = normalize(cross(ballBuffer[id.x].zBasis, ballBuffer[id.x].xBasis));
```

- 새로 추가되는 Ball 들은 좌우에 각자 1 개씩 생성 된다
- 부모 공의 모델 좌표계 값을 기준으로, 랜덤한 값을 가지게 한다
- 기존 부모 공의 방향과 비슷하면서도, 반드시 좌우로 퍼져 나가는 자식 공을 만든다
- 이후 자신의 속도를 기준으로 모델 좌표계 설정

3. 구현 내용과 구현 방법

B. Plane 생성

- 공의 궤적을 나타내는 Plane 을 Procedural Mesh 생성
- Ball 객체 한 개당 Plane 이 한 개씩 존재
- Ball의 경로를 똑같이 따라가도록 한다
- 실제로 공이 날아가는 듯한 느낌을 표현
- Ball이 빠를수록 Plane 의 길이가 길어진다
  - Plane 의 길이를 조절할 수 있게 한다

B-1. C# 스크립트

```
// Plane 의 한 정점 구조체
참조 2개
struct Vertex
{
    public Vector3 position;
    public Vector3 normal;
}
```

```
//plane 을 이루는 정점들을 모아둔 배열
Vertex[] vertexArray;

//vertexArray 를 GPU에 넘기는 버퍼
ComputeBuffer vertexBuffer;

// vertex buffer 실제 길이
int vertexBufferSize;
```

1. Plane 을 구성하는 Vertex 구조체
   - Plane 은 결국 정점들로 구성
   - Quad 단위로 Plane 을 구성
   - 하나의 Plane 은 quad 개수 * 6 개 정점으로 구성된다
   - Plane 을 구성하는 정점 데이터를 배열에 저장하고, 버퍼로 넘긴다

```
//Compute Shader -> ball 커널 함수를 실행시켜, 그 결과를 버퍼에 저장
//ball 들의 위치, 속도 등의 정보를 계산, 저장한다.
computeShader.Dispatch(ballKernelHandleMove, groupSizeX, 1, 1);

//움직인 공에 대한 plane 을 계산한다
computeShader.Dispatch(planeKernelHandleAttach, groupSizeX, 1, 1);
```

```
/*
카메라가 씬을 렌더링 한 후에 자동으로 호출되는 함수
사용자가 자신의 오브젝트를 렌더링 하는 경우에 사용, 나의 경우 Procedural Mesh 를 그리기 위해 사용
ball 이 그려진 후에, plane 을 그리도록 한다
*/
Unity 메시지 | 참조 0개
void OnRenderObject()
{
    //plane 을 그리는 material 을 이용, shader 에서 pass 0 을 사용한다?
    //어차피 renderPlane shader 내에서 pass 는 하나만 존재한다
    planeMaterial.SetPass(0);

    //이번에는 primitive 가 삼각형이라고 넘겨주네 => 버퍼에서 알아서 3개씩 끊어서 읽고
    //1개의 mesh instance = quad 를 그리니까 총 6개의 정점마다 끊어주고
    //하나의 Plane 에는 quadNum 개의 quad 가 존재하므로, 6 * quadNum 개의 정점마다 끊어주어야 한다
    //endIndex = 총 그리는 procedural geometry 의 개수
    Graphics.DrawProceduralNow(MeshTopology.Triangles, 6 * quadNum, endIndex);


    //경계선인 line 을 그린다
    lineMaterial.SetPass(0);

    //하나의 line 은 2 개의 vertex 로 구성되어 있으며
    //경계선은 직육면체로 구성됨 => 12 개의 선으로 구성되어 있다
    Graphics.DrawProceduralNow(MeshTopology.Lines, 2, 12);
}
```

2. Update() 함수 내에서

- Ball 의 위치를 먼저 계산한 뒤, Plane 을 각자의 공에 붙이는 방식으로 진행

3. Plane 의 렌더링

- Unity 엔진에서는 카메라의 씬을 모두 렌더링 한 후에, Procedural Mesh 를 렌더링
- Quad 개수 * 6 개의 정점 단위로 Plane 을 그림

B-2. Compute Shader

```
/*
이후 Vertex Buffer 를 채운다
*/
//현재 ball 에 해당하는 vertex buffer 인덱스를 가져온다
int index = id.x * 6 * quadNum;

//Vertex Buffer 값 초기화
for(int i = 0; i < 6 * quadNum; i++)
{
    //일단 모든 Plane의 정점의 위치는 현재의 위치로
    vertexBuffer[index + i].position = ballBuffer[id.x].position;

    //각 정점의 노멀 방향은 새로 만든 공의 yBasis 방향을 가지도록 한다
    vertexBuffer[index + i].normal = ballBuffer[id.x].yBasis;
}
```

Plane 의 정점들의 데이터를 초기화 하는 코드

자신이 따르는 Ball 의 위치값으로 초기화
Ball 의 모델좌표계의 y 축을 노멀 벡터로 초기화

1. 각 Thread 에서 자신의 Plane 정점에 접근하는 방법

- Plane 의 정점 중 가장 첫번째 인덱스에 먼저 접근한다
  - Id * 6 * quadNum
- 이후 Plane 을 이루는 정점의 개수가 6 * quadNum 개 이므로, 루프를 통해서 모든 정점에 접근

```
//Plane 이 공에 붙어 있을 수 있도록, Plane 의 Vertex 를 갱신하는 커널 함수
//vertex 들의 위치 벡터와 노멀 벡터를 모두 갱신해준다
[numthreads(GROUP_SIZE, 1, 1)]
void attachPlane (uint3 id : SV_DISPATCHTHREADID)
{
    //담당 스레드가 그리려는 공보다 높은 숫자이면, 계산하지 않는다
    if( id.x > (uint)endIndex ) return;

    //일단 현재 담당하는 plane 에 대응되는 공을 가져온다
    Ball ball = ballBuffer[id.x];

    //해당 공에 붙는 Plane 에 해당하는 Vertex Index 를 가져온다
    int index = id.x * 6 * quadNum; //첫번째 인덱스


    /*
    우선 가장 맨 앞에 있는 quad 의, 가장 앞에 있는 Vertex 처리

    맨 앞에 있는 vertex => Position = 항상 공에 붙어 있도록 한다
                           Normal = 공의 yBasis 방향과 같은 방향

    하나의 quad => 2 개의 Triangle 로 구성
    0,4   1

    3     2,5    의 순서를 가진다고, 내가 가정

    따라서 가장 앞에 있는 Vertices 는 0,4 와 1 번째 점들이다
    */
    // top-left
    vertexBuffer[index].position = ball.position - ball.xBasis * halfSize; //공의 반지름 만큼 이동
    vertexBuffer[index+4].position = vertexBuffer[index].position;

    vertexBuffer[index].normal = ball.yBasis;
    vertexBuffer[index+4].normal = ball.yBasis;

    // top-right
    vertexBuffer[index+1].position = ball.position + ball.xBasis * halfSize; //공의 반지름 만큼 이동
    vertexBuffer[index+1].normal = ball.yBasis;
```

2. Plane 이 공을 따라가게 하는 커널 함수

- 해당 Thread id 가 관리하는 Plane 의 첫번째 인덱스에 접근
- 모든 quad 의 정점들이

  0,4 1

  3 2,5

  의 순서를 가진다고 가정
- 첫번째 quad 의 앞 정점들은 항상 공과 붙어있다

```
//이후, 뒤에서 따라가는 Vertices 의 위치와 노멀을 계산해야 하는데 ...
/*
먼저, 맨 앞 quad 에서 => 공의 위치와 뒤 Vertices 의 중심을 비교한다

둘의 거리 차이가 일정 값, threshold 값을 넘길 때 나머지 Vertices 의 값을 갱신하도록 한다
기본 threshold => planeLength 변수 값으로 받는다

threshold 값은 공의 속력 값에 비례하도록 한다
    그래서 속력이 클수록, threshold 가 커지게 해, 각 quad 가 길어지게 한다
    속력이 빠를 수록 plane 이 길게 나타나게 된다
*/
//공과 첫번째 quad 뒤 정점의 중심 사이 거리
float3 backPos = (vertexBuffer[index+3].position + vertexBuffer[index+2].position)/2;
//그 거리가 일정 값 이하라면, 다른 Vertices 의 값을 갱신하지 않는다
if(length(ball.position - backPos) < ball.speed * planeLength) return;


//맨 앞의 quad 에서 앞 뒤 간격이 threshold 보다 커져서, 갱신을 하게되는 경우
/*
정점들이 앞에 있던 정점을 따라 가야 한다, 앞에 있던 정점이 가지고 있던 값을 가져야 한다

뒤에 있는 정점들부터 먼저 앞 정점의 값을 가지게 하고

맨 앞 quad에 대해서 마지막에 갱신
    첫번째 quad 의 뒤 2개의 정점이 앞 2개의 정점 값을 가지게 한다
    결과적으로 보면 첫번째 quad 가 접히는 결과

노멀 벡터도 그대로 이어 받으면 되겠지?
*/
//끝에 있는 quad 를 가리키도록 index 를 수정한다
index += 6 * (quadNum - 1);
```

- 공과 첫번째 quad 의 앞 정점의 중심 사이의 거리를 체크
- 해당 거리가 일정 거리 이상일 때만 Plane 이 이동
- 뒤의 Plane 의 정점들은 앞의 정점들의 위치로 이동하는 방식으로, Plane 전체가 이동하게 된다
- 앞 정점의 노멀벡터 값도 그대로 넘겨 받는다

```
//끝에 있는 quad 부터 첫번째 quad 를 향하도록 반복문을 진행한다
for(int j = quadNum; j > 1; j--) {
    /*
    -6    -5

    0,4   1

    3     2,5
    */
    // bottom-left
    vertexBuffer[index+3].position = vertexBuffer[index].position;
    vertexBuffer[index+3].normal = vertexBuffer[index].normal;


    // bottom-right
    vertexBuffer[index+2].position = vertexBuffer[index+1].position;
    vertexBuffer[index+5].position = vertexBuffer[index+1].position;

    vertexBuffer[index+2].normal = vertexBuffer[index+1].normal;
    vertexBuffer[index+5].normal = vertexBuffer[index+1].normal;


    // top-left
    vertexBuffer[index].position = vertexBuffer[index-6].position;
    vertexBuffer[index+4].position = vertexBuffer[index-6].position;

    vertexBuffer[index].normal = vertexBuffer[index-6].normal;
    vertexBuffer[index+4].normal = vertexBuffer[index-6].normal;


    // top-right
    vertexBuffer[index+1].position = vertexBuffer[index-5].position;
    vertexBuffer[index+1].normal = vertexBuffer[index-5].normal;

    //quad 를 하나씩 줄여나간다
    index -= 6;
}
```

```
    // 다시 맨 앞 quad 까지 왔다
    // bottom-left
    vertexBuffer[index+3].position = vertexBuffer[index].position;
    vertexBuffer[index+3].normal = vertexBuffer[index].normal;

    // bottom-right
    vertexBuffer[index+2].position = vertexBuffer[index+1].position;
    vertexBuffer[index+5].position = vertexBuffer[index+1].position;

    vertexBuffer[index+2].normal = vertexBuffer[index+1].normal;
    vertexBuffer[index+5].normal = vertexBuffer[index+1].normal;
}
```

- Plane 의 가장 뒤에 있는 정점에서부터, 앞에 있는 정점의 위치와 노멀벡터 값을 가진다
- 가장 앞에 있는 quad 까지 처리하면, Plane 이 이동한 효과를 낼 수 있다

3. 구현 내용과 구현 방법

C. Ball 렌더링

- 계산된 Ball 의 위치와 좌표계에 맞춰서, 적절한 위치로 이동시키고 회전
- 기본적인 블린-퐁 라이팅 모델 계산
- 정해진 영역 내에서 뒤로 갈수록, 카메라로부터 멀어질수록 색깔이 어두워지게 한다
- Rim Lighting 구현, 테두리를 검정색 선으로 강조한다

C-1. Vertex Shader

```
//Vertex Shader
                    //모델 내에서 Vertex 의 id          그려지는 여러 모델 중, 현재 모델의 id
v2f vert (appdata v, uint vertexID : SV_VertexID, uint instanceID : SV_InstanceID)
{
    v2f o;

    /*
    주어진 Ball 의 크기를 바꿀 수 있다
    입력받은 radius 값을 이용
    정점을 normal vector 방향으로 이동시키면
    중심으로 부터 멀어져 => 공이 커지는 효과를 낼 수 있다

    모델 공간 내에서, radius 만큼 늘어난 정점의 위치를 구함
    */
    //Vertex Extrude
    v.vertex = float4(v.vertex.xyz + v.normal * radius, 1);


    /*
    SV_InstanceID 값을 이용
    현재 모델에 해당하는 Ball 객체를 찾을 수 있다

    해당 Ball 의 모델 좌표계를 가져온다
    모델 좌표계 => 공이 움직이는 방향에 맞춰져서 설정
    월드 공간 좌표계를 기준으로 값을 가지고 있다
    */
    float3 xaxis = ballBuffer[instanceID].xBasis;
    float3 yaxis = ballBuffer[instanceID].yBasis;
    float3 zaxis = ballBuffer[instanceID].zBasis;
    float3 pos = ballBuffer[instanceID].position; //현재 공의 위치
```

- Vertex Extrude
  - 정점 노멀 벡터 방향으로 정점을 이동시켜, 그리고자 하는 Mesh 의 크기를 키운다
  - 앞서 계산한 모델 좌표계 벡터와 위치를 가져온다
  - 위의 값은 모두 월드 좌표계를 기준으로 측정된 값

```
/*
변환 후 좌표계의 벡터를 알고 있으니, 변환행렬을 구할 수 있다
이 행렬 자체가 모델 -> 월드 변환 행렬
회전 + 이동 변환
*/
float4x4 motionMat = float4x4 (
                              xaxis.x, yaxis.x, zaxis.x, pos.x,
                              xaxis.y, yaxis.y, zaxis.y, pos.y,
                              xaxis.z, yaxis.z, zaxis.z, pos.z,
                              0, 0, 0, 1
                             );
//회전 변환 부분만 따로 구함
float3x3 L = float3x3 (
                       xaxis.x, yaxis.x, zaxis.x,
                       xaxis.y, yaxis.y, zaxis.y,
                       xaxis.z, yaxis.z, zaxis.z
                      );

//정점의 위치를, 모델 공간에서 월드 공간으로 변환
v.vertex = mul(motionMat, v.vertex);


/*
정점의 위치가 변하면, normal vector 도 변환해주어야 한다
노멀벡터는 월드 변환의 [L | t] 에서 이동변환을 제외한 L 에 대해서만 생각
=> normal 의 경우엔 inverse trasnpose 를 이용하여 변환한다. 즉 (L^-1)^T
그런데, 이 변환의 경우 비균등 확대축소 없이, 그냥 이동,회전 만 있는 강체변환 이니까,
변환을 노멀에 그대로 적용해도 된다
그럼 노멀 벡터도 월드 공간으로 변환이 이루어 졌다
*/
v.normal = mul(L, v.normal);
```

- 모델 변환 적용
  - 모델 좌표계의 벡터를 알고, 또 모델의 위치를 아는 상황
  - 월드 좌표계 -> 모델 좌표계 일치시키는 변환이 모델 변환
  - 모델의 정점을 월드 좌표계로 변환
  - 선형 변환 부분만 이용해서 정점의 노멀 벡터도 월드 좌표계로 변환한다

```
    // 월드 공간에서 좌표를 구한다
    //o.vertexWorld = mul(unity_ObjectToWorld, v.vertex);
    o.vertexWorld = v.vertex;

    // 클립 공간에서 좌표를 구한다
    //o.vertexClip = UnityObjectToClipPos(v.vertex);
    o.vertexClip = UnityWorldToClipPos(v.vertex);

    // 월드 공간에서 노멀벡터를 구한다
    //o.normalWorld = v.normal;
    o.normalWorld = v.normal;

    return o;
}
```

- 정점의 월드 좌표계에서 좌표
- 정점의 클립 공간에서의 좌표
- 정점의 월드 좌표계에서 노멀 벡터
- 값들을 저장 후, 레스터라이저로 넘긴다

```
fixed4 frag (v2f i) : SV_Target
{
    //해당 fragment 의 월드공간에서 노멀 벡터를 구한다
    float3 norVec = normalize(i.normalWorld); //normalize 필수, 보간 과정에서 정규화를 기대할 수 없다

    //뷰 벡터 = 해당 프레그먼트에서 카메라를 바라보는 벡터
    //뷰 벡터를 월드 공간 좌표계 기준 값으로 구한다 (프레그먼트의 월드 공간 좌표를 넣어서 구할 수 있다)
    float3 veiwVec = normalize(UnityWorldSpaceViewDir(i.vertexWorld));

    //빛 벡터 = 해당 프레그먼트에서 광원을 바라보는 벡터
    //월드 공간에서 빛 벡터를 구한다
    float3 lightVec = normalize(UnityWorldSpaceLightDir(i.vertexWorld));


    //해당 Fragment 의 월드 공간에서의 벡터 값을 이용해서 Lighting 을 계산한다
    // 1. 난반사, 디퓨즈
    // 노멀 벡터와 빛 벡터를 이용하면, 프레그먼트에 들어오는 빛의 양을 구할 수 있다
    float lightAmount = max(_Ambient, dot(norVec, lightVec));
    float4 diffuseTerm = lightAmount * _Color * _LightColor0; //물체의 색과 빛의 색 고려


    // 2. 스펙큘러, 정반사 + 간접광
    //빛 벡터를 반사시켜서, 프레그먼트에서 반사되어져서 나오는 빛 반사 벡터를 구한다
    float3 reflectVec = reflect(lightVec, norVec);

    //빛 반사 벡터와, 뷰 벡터 사이 내적을 구해 눈으로 들어오는 빛의 양을 구한다
    //최소 일정 간접광이 들어온다
    float3 lightAmountOnEye = max(_Ambient, dot(veiwVec, reflectVec));

    //눈으로 들어오는 빛의 양이 줄어들면, 반사광에 의한 색은 기하급수적으로 줄어든다
    //매끄러움 을 통해 이를 조절할 수 있다
    float3 specular = pow(lightAmountOnEye, _Shininess);
    float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0;


    // 3. 최종 색깔
    float4 finalColor = diffuseTerm + specularTerm;
```

C-2. Fragment Shader

- 기본적인 블린-퐁 모델 적용

```
/*
뒤로 갈수록 어둡게 만들고 싶다

프레그먼트의 z 위치에 따라서 0~1 값을 구한다
그 값을 이용해서, z 위치가 앞에 올수록 더 밝게 보이게 한다
*/
//smoothsteop => 부드러운 보간
float fragZ = smoothstep(-limitSize.z, limitSize.z, i.vertexWorld.z) * 0.5;
// 가까울 수록 밝게 보이고 싶다
finalColor.xyz *= (1 - fragZ);
```

- 카메라로부터 멀어질수록 어두워지는 색깔 구현
  - 프레그먼트의 월드 공간에서 좌표 값을 이용
  - Smoothstep 함수를 이용, 이동가능 공간 내에서 해당 프레그먼트의 상대적인 위치 값을 얻어낸다
  - 해당 값을 이용해서 밝기 조절

```
/*
Rim Lighting
    |
프레그먼트의 월드 공간에서 노멀벡터와 뷰 벡터 내적을 구한다
가장자리, Rim 에 해당할수록 그 값이 낮게 나올 것이다
    |   잘 안보이는 fragment 이니까
    |
그걸 역으로 이용하면, 가장자리만 다른 색을 칠할 수 있다
*/
float rimAmount = max(0.0, dot(norVec, veiwVec));
//가장자리로 판단되면, 그냥 검정색만 리턴시킨다
if(rimAmount < 0.4f) finalColor = float4(0,0,0,1);
```

- Rim Lighting 구현
  - 월드 좌표계에서 프레그먼트의 노멀 벡터와
    프레그먼트에서 카메라를 바라보는 뷰 벡터의 내적을 이용
  - 눈에 잘 보이지 않는 영역을 가장자리로 판단
  - 검정색을 가지게 해, 테두리를 확실히 볼 수 있게 한다

3. 구현 내용과 구현 방법

D. Plane 렌더링

- 끝으로 갈수록 투명해지도록 해서, 실제로 날아가는 듯한 효과를 준다
- Culling Off
  - Quad 로 구성
  - 뒤집어져도 볼 수 있어야 하므로 Culling 기능을 끈다
- Alpha Blending
  - 투명하게 보이기 위해서 알파값을 이용
  - 따라서 Alpha Blending 기능을 활성화해야 한다

D-1. Shader

```
Tags
{
    //알파값을 가지고 사용하기 때문에
    //Transparent 큐 를 사용 => 가장 마지막에 그린다
    "Queue" = "Transparent"
}


//알파 블렌딩
//가장 기본적인 알파 블렌딩을 진행한다
//생성되는 색깔 * 생성되는 알파 + 기존 뒤에 있던 색깔 * (1-알파)
Blend SrcAlpha OneMinusSrcAlpha


//Quad 를 언제나 볼 수 있도록 Culling 을 끈다
//뒤집혀도 렌더링을 진행한다
Cull Off
```

- Unity ShaderLab 의 기능 활용
  - Alpha Blending 설정
  - Culling 기능 끄기

D-2. Vertex Shader

```
//Vertex Shader
v2f vert (uint vertex_id : SV_VertexID, uint instance_id : SV_InstanceID)
{
    v2f o;

    /*
    먼저 전체 버퍼에서 처리하려는 정점 인덱스를 찾는다

    각 Plane 은 SV_InstanceID 로 구별되고
    하나의 인스턴스는 6 * quadNum 개의 정점으로 이루어져 있다
    그리고 하나의 인스턴스 내에서 정점의 id 는 SV_VertexID 로 나타난다
    */
    int index = instance_id * 6 * quadNum + vertex_id;


    //전체 quad 에서 가장 top 에 있는 정점과 가장 bottom 에 있는 정점을 가져온다
    int topLeftIndex = instance_id * 6 * quadNum;   // most top left
    int topRightIndex = topLeftIndex + 1;           // most top right

    int bottomRightIndex = (instance_id + 1) * 6 * quadNum -1; // most bottom right
    int bottomLeftIndex = bottomRightIndex - 2;                // most bottom left


    // 해당 plane 의 most top 의 중점 위치
    float3 topPos = ( vertexBuffer[topLeftIndex].position
                    + vertexBuffer[topRightIndex].position ) / 2;

    // plane 의 most bottom 의 중점 위치
    float3 bottomPos = ( vertexBuffer[bottomLeftIndex].position
                       + vertexBuffer[bottomRightIndex].position ) / 2;
```

- 정점의 상대적 위치 계산
  - 전체 Vertex 버퍼에서, 해당 정점에 대응되는 인덱스를 계산
  - 해당 정점이 포함된 Plane 의 가장 앞 위치와 가장 뒤 위치를 구한다

```
    // 전체 plane 의 길이
    float planeLength = length(topPos - bottomPos);

    // topPos 에서 현재 정점 까지의 거리
    float topToVert = length(topPos - vertexBuffer[index].position);


    //전체 plane length 에 대해서, 현재 정점의 상대적인 위치를 구하고 싶다
    //smoothstep 을 이용해, 보간을 통해서 현재 정점의 상대적 위치를 구한다
    float ratio = smoothstep(0, planeLength, topToVert);


    //버퍼에서 가져온 현재 정점의 위치 = 월드 공간 기준 좌표계 값
    //바로 클립 공간 위치롤 변환시켜준다
    o.vertex = UnityWorldToClipPos(float4(vertexBuffer[index].position, 1.0f));

    //버퍼에서 값을 바로 가져와 => 월드 공간에서 정점의 값을 채운다
    o.vertexWorld = float4(vertexBuffer[index].position, 1.0f);
    o.normalWorld = vertexBuffer[index].normal; // 이미 월드 공간에서 노멀

    //앞서 구한 상대적인 위치 값을 넘긴다 => 레스터라이저에 의해 보간
    o.ratio = ratio;

    return o;
}
```

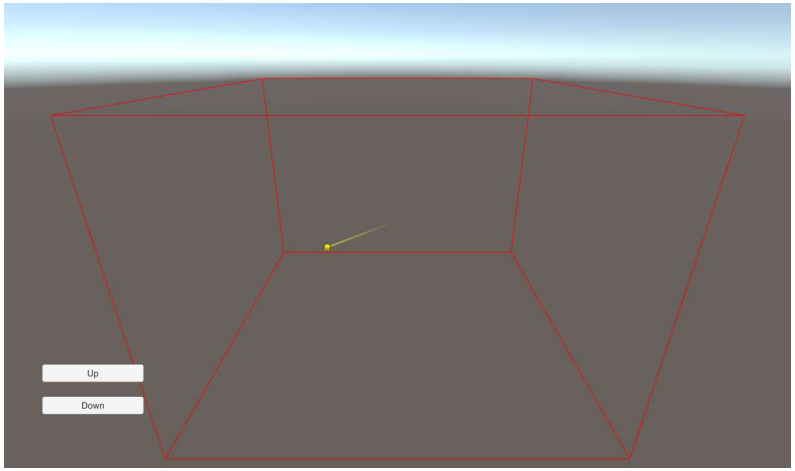
- 전체 Plane 내에서
  현재 정점의 상대적인 위치를
  smoothstep 함수를 통해서
  구한다
- 상대적 위치 값인 ratio 를
  레스터라이저롤 넘긴다

D-2. Fragment Shader

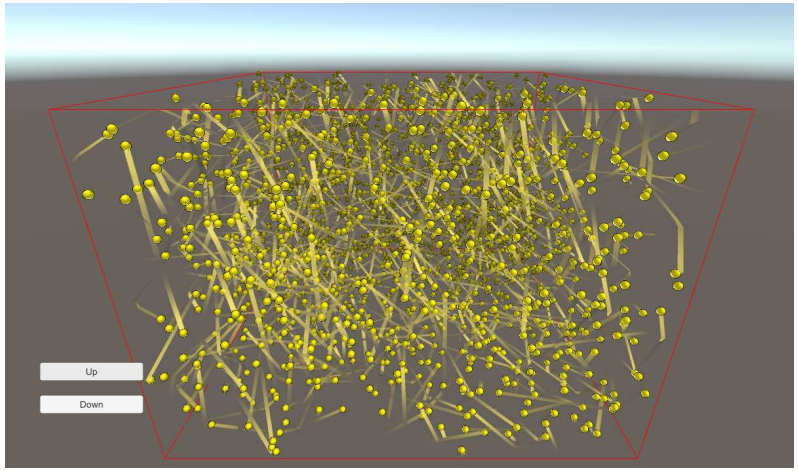
```
//Fragment Shader
fixed4 frag (v2f i) : SV_Target
{
    // 월드 공간에서 노멀 벡터
    float3 norVec = normalize(i.normalWorld);

    // 월드 공간에서 빛 벡터 => fragment 에서 광원을 바라보는 벡터를 구한다
    float3 lightVec = normalize(UnityWorldSpaceLightDir(i.vertexWorld));

    //해당 fragment 에 들어오는 빛의 양을 구한다, 최소 0.2 값은 들어 오게 한다
    float lightAmount = max(0.2, dot(lightVec, norVec));


    //기존 Plane 의 색깔 + 앞 부분에서 상대적으로 멀어질수록 투명해지게 한다
    float4 modiColor = float4(_Color.x, _Color.y, _Color.z, 1 - i.ratio);

    //최종 색깔
    fixed4 finalColor = modiColor * lightAmount * _LightColor0;

    return finalColor;
}
```

- 해당 Fragment 에 들어오는 빛의 양 계산
- 앞서 계산한 Ratio 값을 이용
- Plane 의 끝에 위치할수록 투명한 색을 가지도록 한다

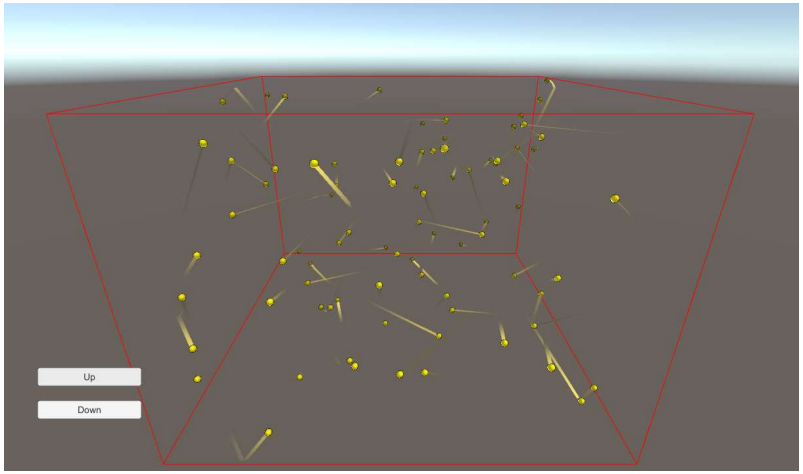
4. 결과 (https://youtu.be/-stKSOqev48)



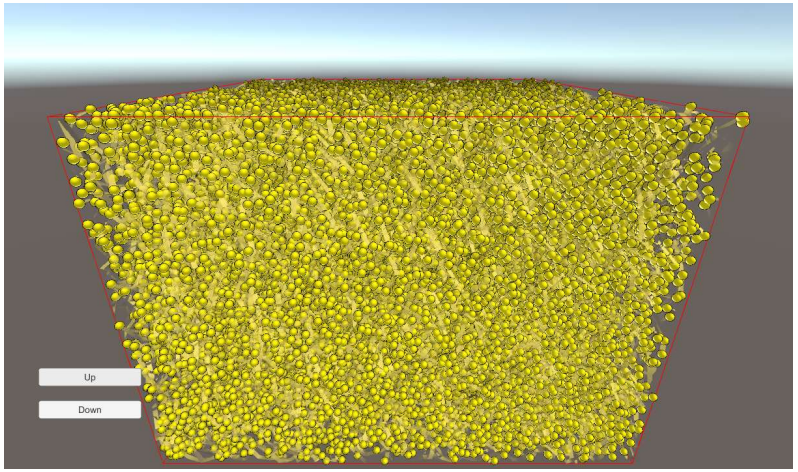
1 개



2187 개



81 개



50000 개

- 대략 20000 개 까지는 무리없이 동작하는 것을 확인
- 50000 개 부터는 끊김이 너무 심해서, 더 이상 진행이 불가능했다
- 공의 꼬리를 표현하는 Plane 이 잘 그려지는 것을 확인
- 실행도중 공의 크기를 변화시키거나, Plane 의 길이를 변화시키는 동작이 잘 작동했다

5. 느낀 점

- Compute Shader 사용을 연습해 볼 수 있어 좋았다
- Mesh 를 직접 만들고 렌더링하는 것이 굉장히 어렵고 복잡하다는 것을 알 수 있었다
- 하지만 Mesh 를 직접 만들어야만 표현할 수 있는 것들이 있다는 것을 알 수 있었다