셰이더 프로그래밍을 하면 Uniform 으로 값을 계속 바꿀 수가 있다. 재사용 유리. C++의 클래스처럼 셰이더도 클래스의 원형과 같이 재사용에 좋다.
셰이더 클래스 지정해 두고 멤버변수들의 ID 로 재사용하는 것과 같음. 인스턴스를 만들고 나서 외부에서 제어할 수 있는 것은 원형이 바뀌지 않으면 다시 컴파일하지 않아도 됨.
외부에서 줄수 있는 값은 Attribute 아니면 Uniform으로 만들면 된다. 언리얼엔진에서도 머트리얼 만들어 색깔만 다르다면 Inst 만들어서 쓰는 게 훨씬 좋다. 컴파일 시간 훨씬 줄어든다.

셰이더 프로그래밍

Lecture 4

이택희

파이프라인별로 앞의 출력이 뒤의 입력이 된다. 매번 버텍스 버퍼, 셰이더를 만들다 보니 너무 코드가 길어지고 있다. 어떤 방식으로 넣어야 좋을지 생각할 것이다.
프래그먼트 셰이더만으로 할 수 있는 것이 많다. 간단한 몇 가지 제외하고는 프래그먼트로'만' 할 수 있는 것들 해보겠다. 물결„„? 이거„„? 시험 나올 수 있다. 근데 여태까지 한명만 맞혔다. ㅋㅋ

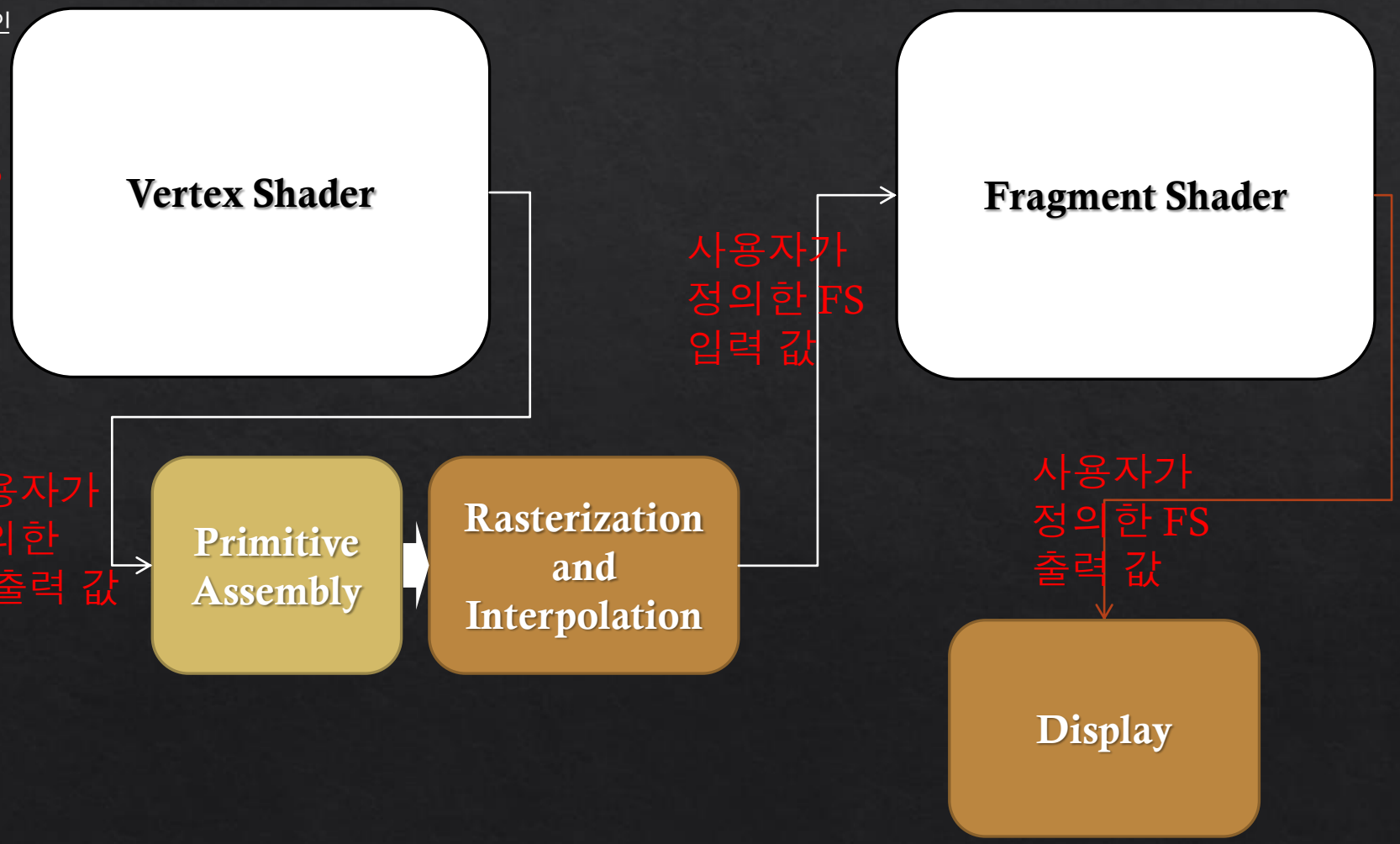
개요

- 프래그먼트 셰이더
- Storage Qualifier
- 버텍스 셰이더 입력 데이터 패킹
- 실습

프래그먼트 셰이더

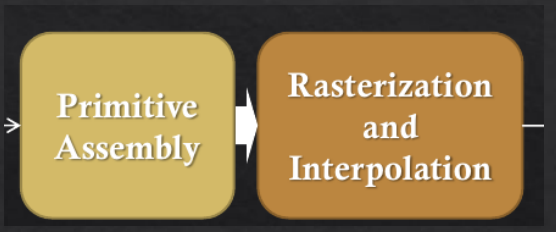
Shader 입출력

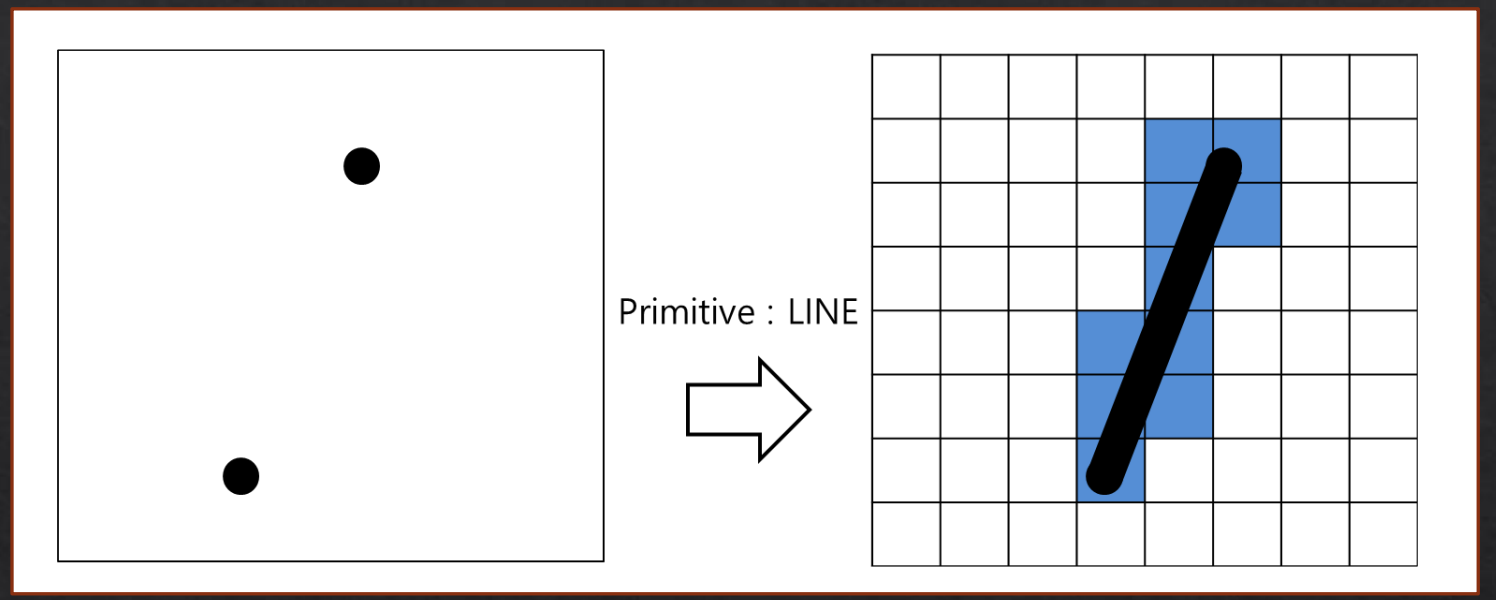
유일하게 받을 수 있는 값: 어트리뷰트?
중간에 새로운 데이터를 받으면 파이프라인 자체가 비효율적으로 됨.
반드시 받아야 할 게 이전 스테이지에 있어야 함.



버텍스에서 받는 버텍스가 두 개더라도 단계를 거치면서 수백개가 될 수도, 수천ㄴ개가 될수도. 입력과 출력의 개수가 다르다.
밑에 그림은 Primitive가 LINE 으로 결정되어 있기 때문에 라인으로 그려져야 한다. 그 과정이 밑의 노란색과 주황색에서 이루어짐.

프래그먼트 셰이더



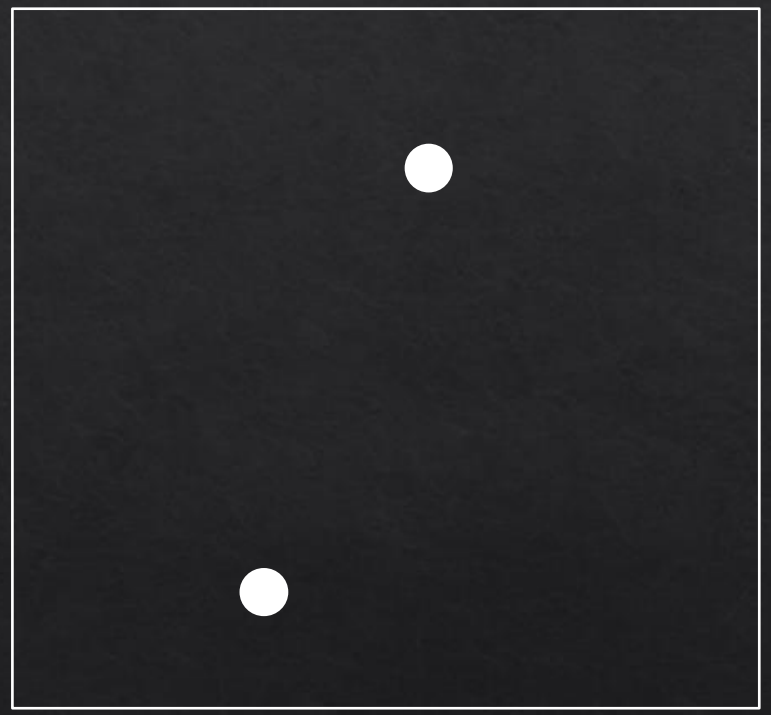


버텍스 셰이더의 출력과 프래그먼트 셰이더 입력의 타입은 서로 같아야 함

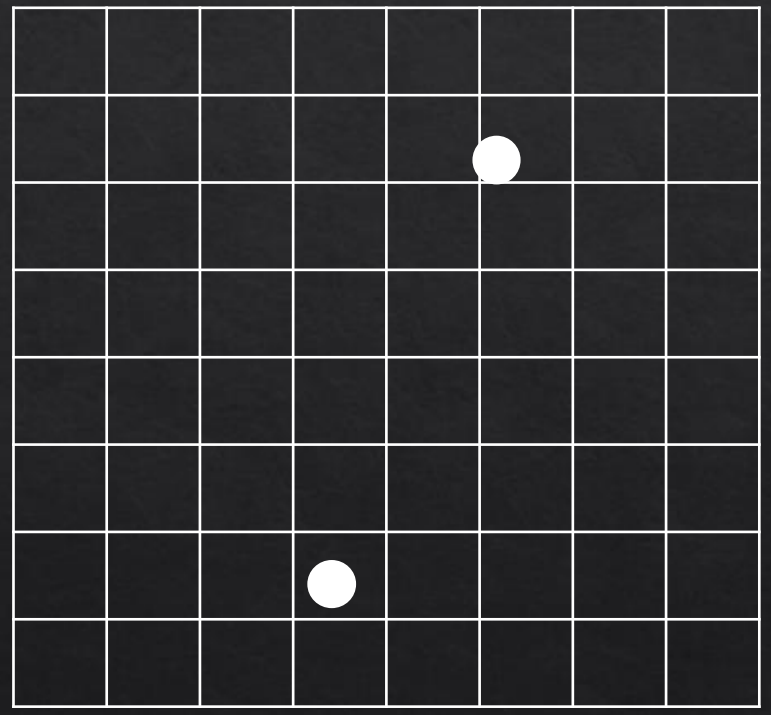
vertex → primitive → fragments 과정에서 따라서 보간이 일어남

프래그먼트 셰이더에서는 이전 과정에서 출력이 안 된 놈을 채우는 기능이 없음.

프래그먼트 셰이더



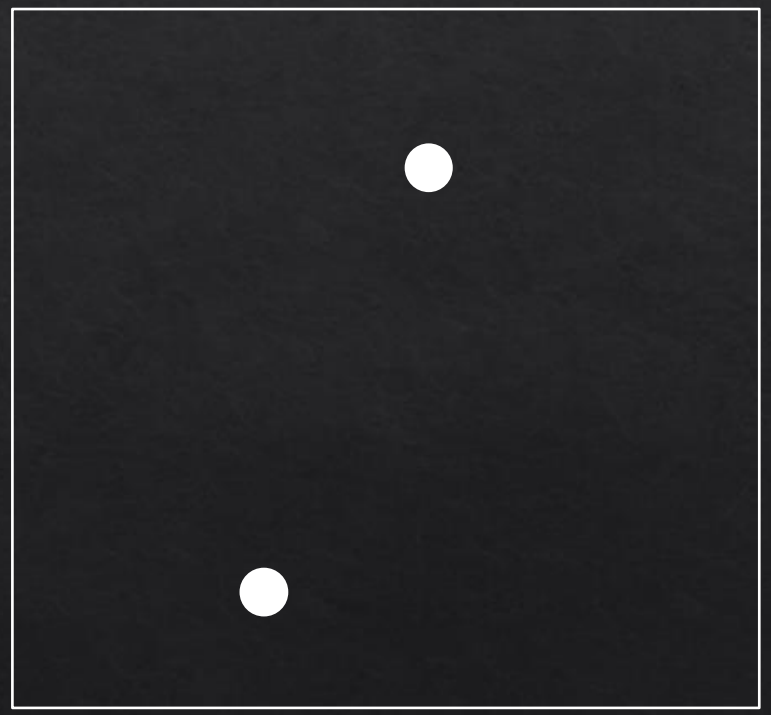
Primitive :
LINE



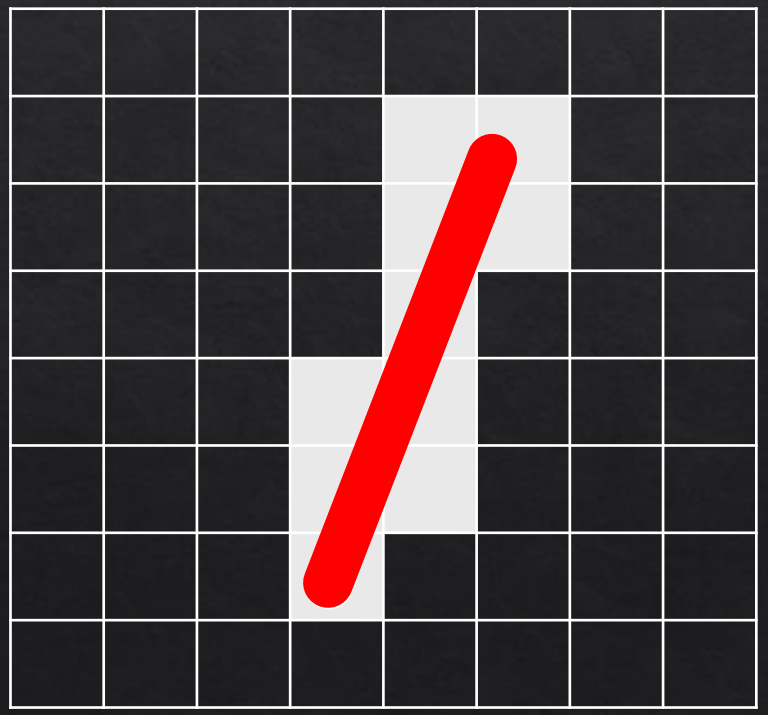
프래그먼트 셰이더

- 버텍스는 두 개이지만 보간된 프래그먼트는 10개

프래그먼트 셰이더는
10개의 버텍스 하나하나를 색 칠하고, 뭐 입히고 이런식으로 해서 출력해준다.
일반적인 케이스에서 버텍스 셰이더보다 프래그먼트 셰이더의 연산이 더 많음.
10만개, 20만개씩 개수가 많아진다면 계산양의 차이가 아주 커짐.
그래서 가급적이면 프래그먼트 셰이더의 계산량을 줄이려고 하는 움직임이 있음.

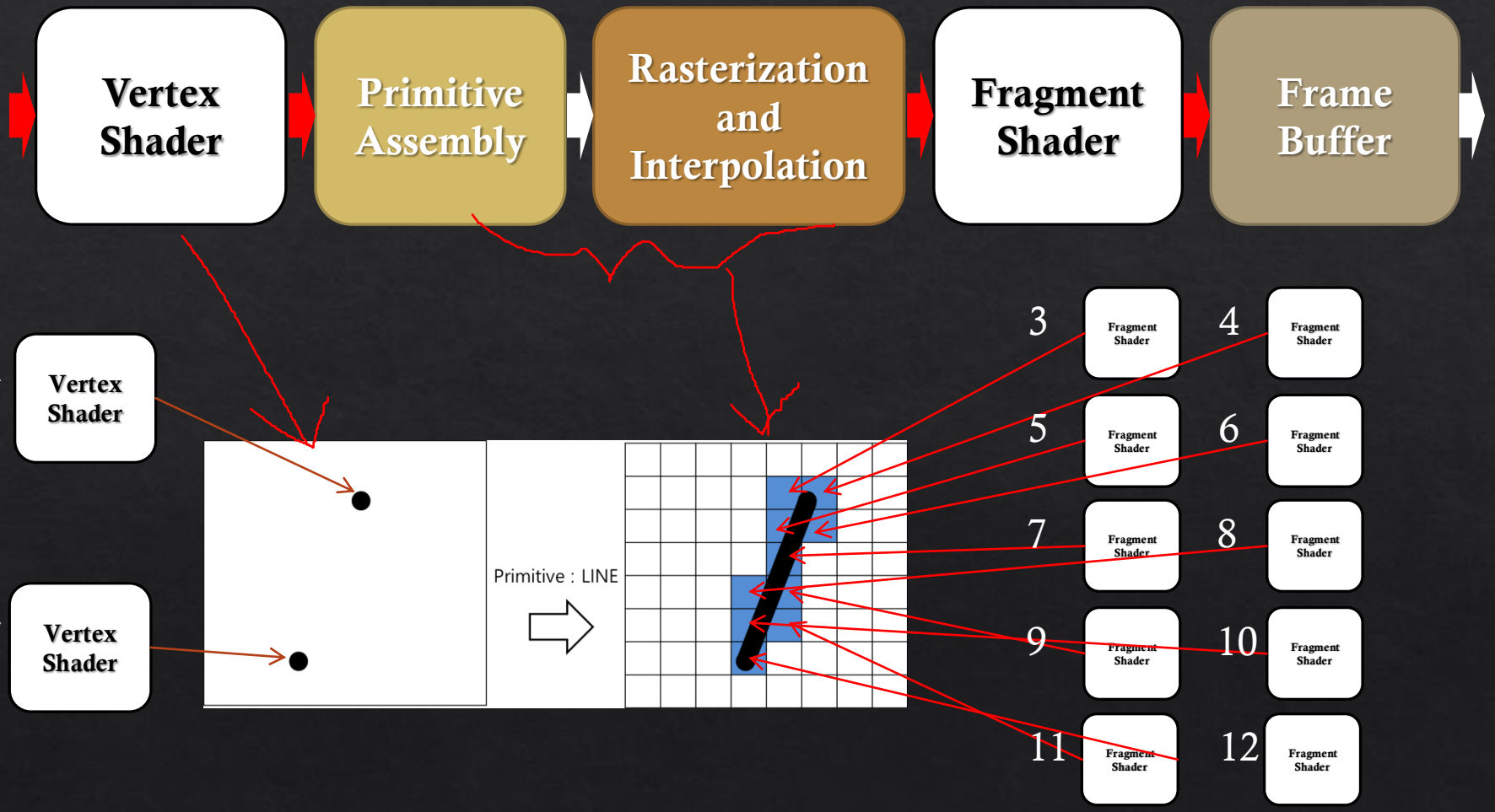


Primitive :
LINE

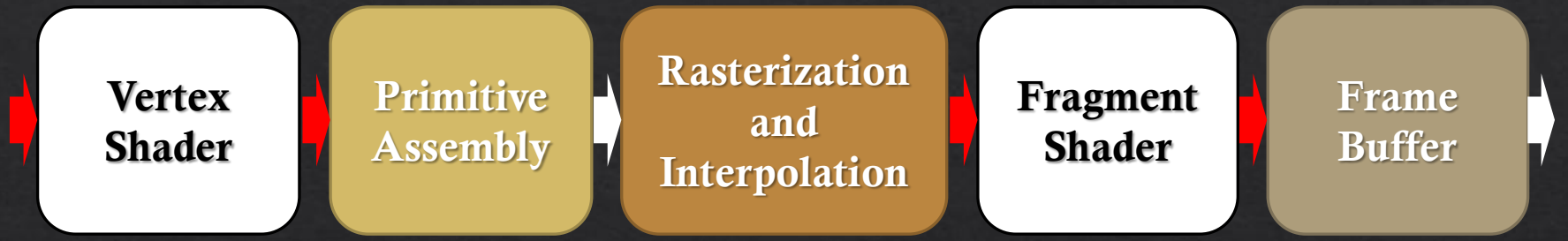


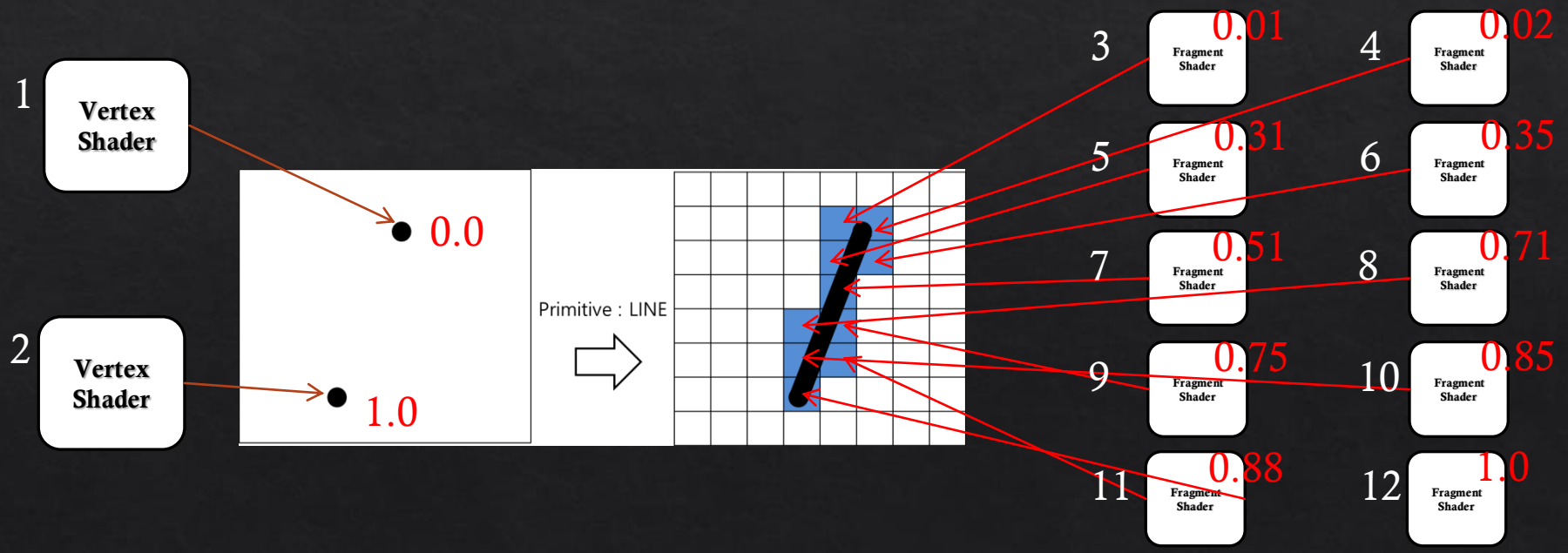
총 10번의 프래그먼트 셰이더가 동작함 !

프래그먼트 셰이더


Vertex Shader
Primitive Assembly
Rasterization and Interpolation
Fragment Shader
Frame Buffer
1
Vertex Shader
2
Vertex Shader
Primitive : LINE
3
4
5
6
7
8
9
10
11
12
Fragment Shader

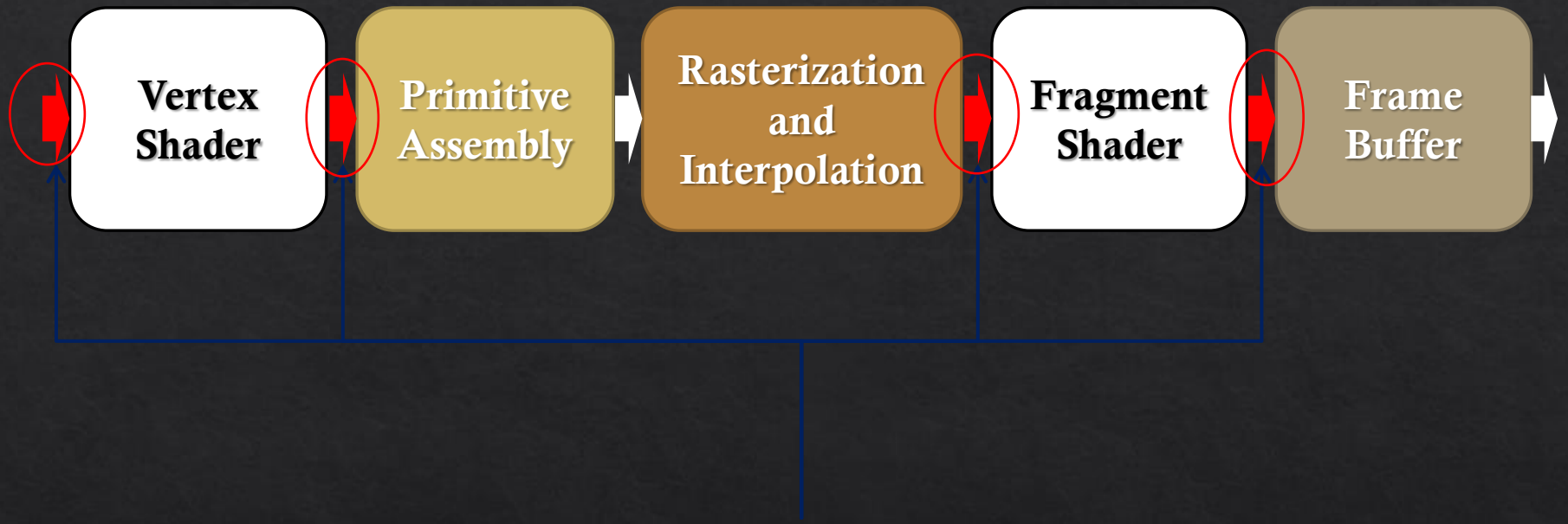
프래그먼트 셰이더


Vertex Shader
Primitive Assembly
Rasterization and Interpolation
Fragment Shader
Frame Buffer


1
Vertex Shader
2
Vertex Shader
0.0
1.0
Primitive : LINE
3
Fragment Shader
0.01
4
Fragment Shader
0.02
5
Fragment Shader
0.31
6
Fragment Shader
0.35
7
Fragment Shader
0.51
8
Fragment Shader
0.71
9
Fragment Shader
0.75
10
Fragment Shader
0.85
11
Fragment Shader
0.88
12
Fragment Shader
1.0

Storage Qualifier

Storage Qualifier



각 Stage 의 입출력을 지정해야 함

동그라미 표시한 곳이 입출력 지정 필요한 곳 ㅇㅇ

버텍ㄱ스 셰이더의 입력이란 옛날에 'in'이라고 했음. in = attribute임. .vs 파일의 in float 어쩌구라고 선언하는 것들은 attribute float 어쩌구라고 선언할 수 있음, 동일함. out은 아직 쓴 적이 없다. varying도 안 씀. out을 varying이라고 적어도 됨. 우리는 in 과 out을 쓸 것이긴 하다.

Storage Qualifier

Stage 자체의 입출력 값에 대해 쓰이는 Qualifier

Storage Qualifier	Meaning
< none: default >	local read/write memory, or an input parameter to a function
const	a variable whose value cannot be changed
in	linkage into a shader from a previous stage, variable is copied in
out	linkage out of a shader to a subsequent stage, variable is copied out
attribute	compatibility profile only and vertex language only; same as **in** when in a vertex shader
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application
varying	compatibility profile only and vertex and fragment languages only; same as **out** when in a vertex shader and same as **in** when in a fragment shader

주의

옆에는 함수 안에서 쓰는 것이고 위에는 파이프라인 상에서 입/출력으로 쓰는 것이다.

Stage 내부에서 쓰이는 Modifier

Access Modifier	Description
`in`	Value copied into a function (default if not specified)
`const in`	Read-only value copied into a function
`out`	Value copied out of a function (undefined upon entrance into the function)
`inout`	Value copied into and out of a function

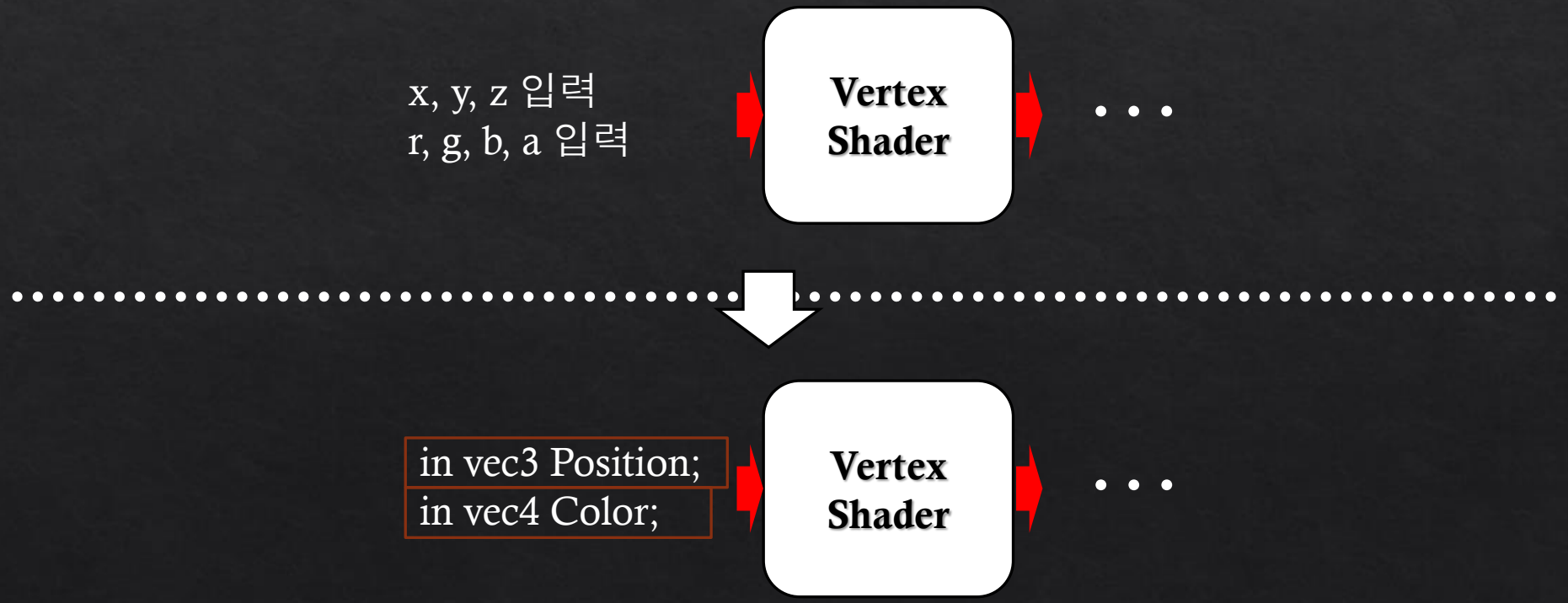
Storage Qualifier(in/out)

- 각 Stage 의 입출력을 지정
  - 그래픽스 파이프라인 외부로의 입출력

  - 그래픽스 파이프라인 Stage 간의 입출력

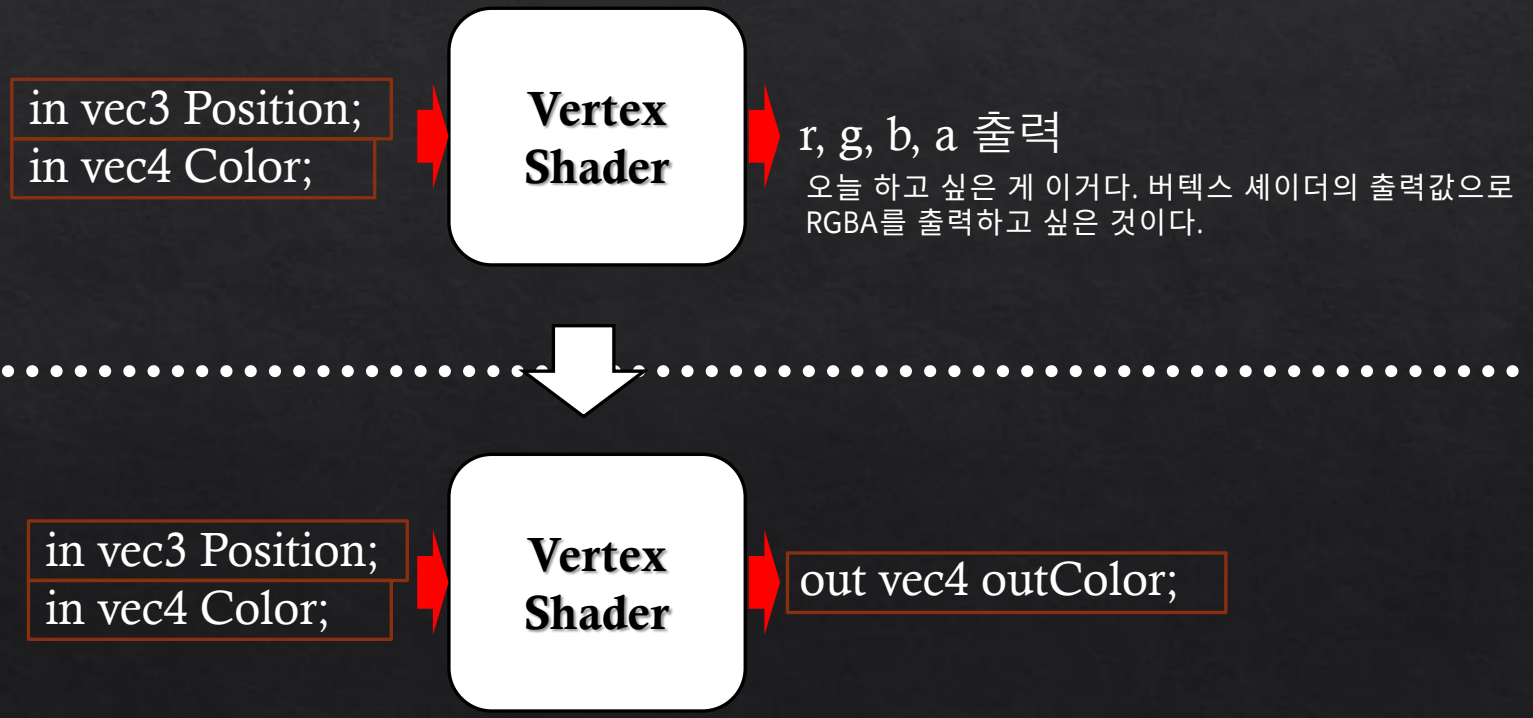
Storage Qualifier(in/out)

- 그래픽스 파이프라인 외부로부터의 입력
  - ◈ 버텍스 셰이더의 입력 값



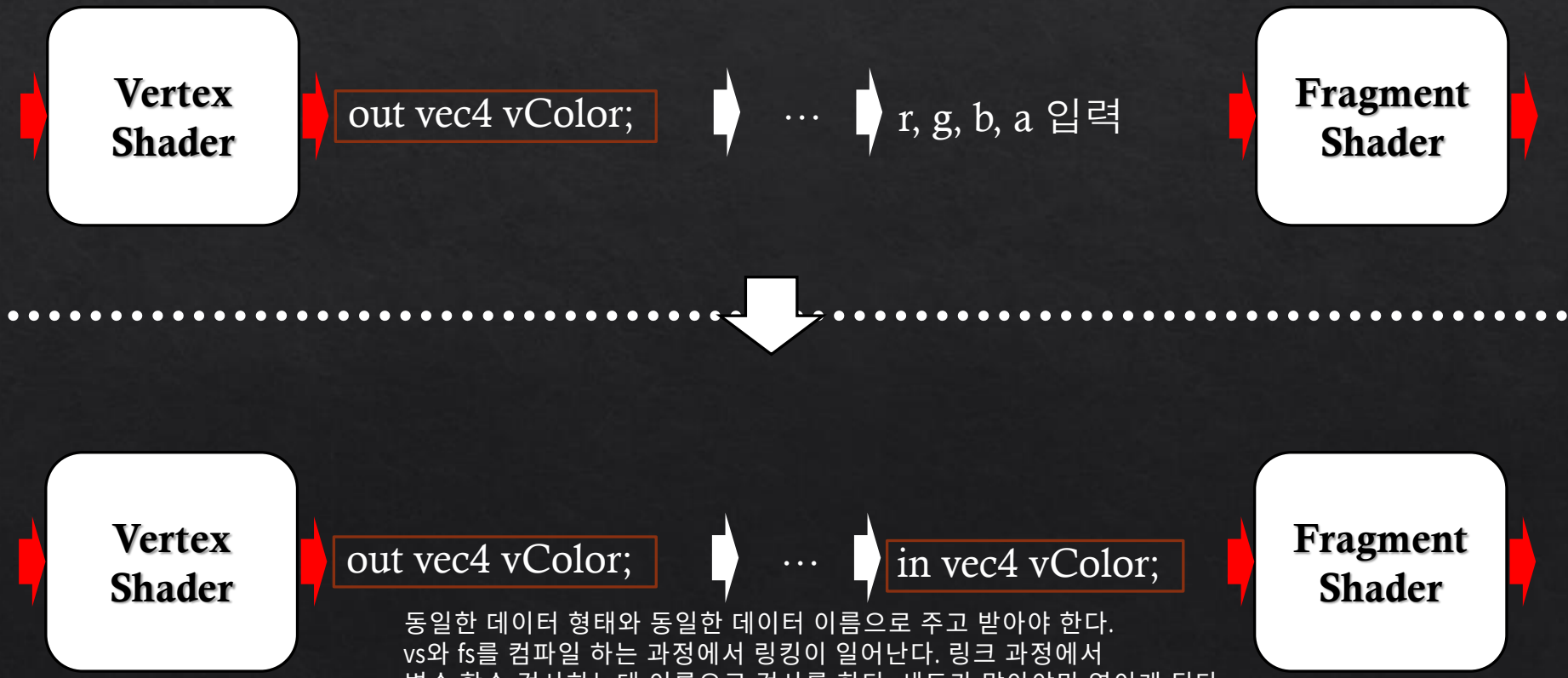
Storage Qualifier(in/out)

- 그래픽스 파이프라인 다음 Stage로 출력
  - ◈ 버텍스 셰이더의 출력 값



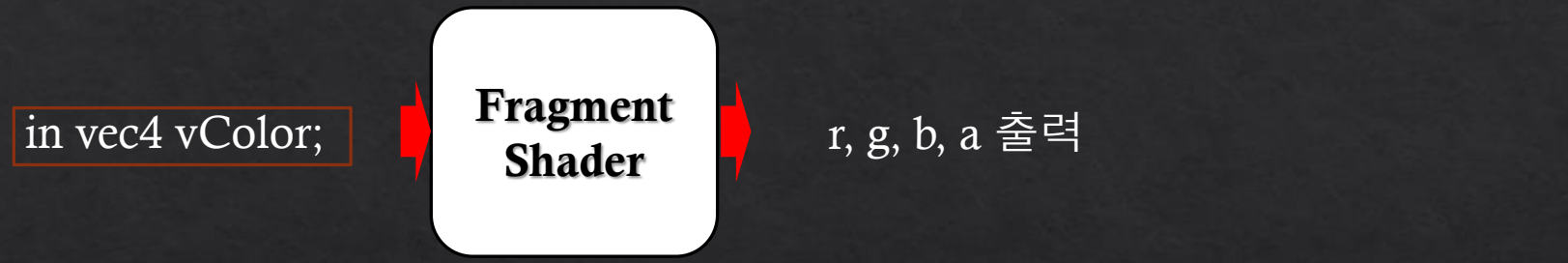
Storage Qualifier(in/out)

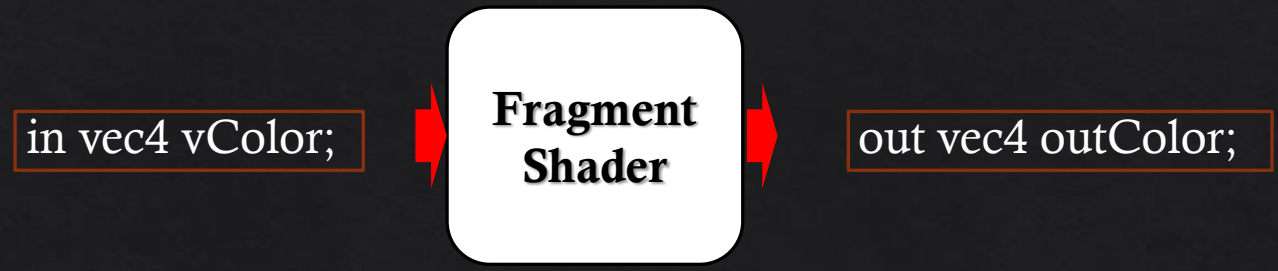
- 그래픽스 파이프라인 이전 Stage 로부터의 입력
  - ◈ 버텍스 셰이더의 출력값 == 프래그먼트 셰이더의 입력값



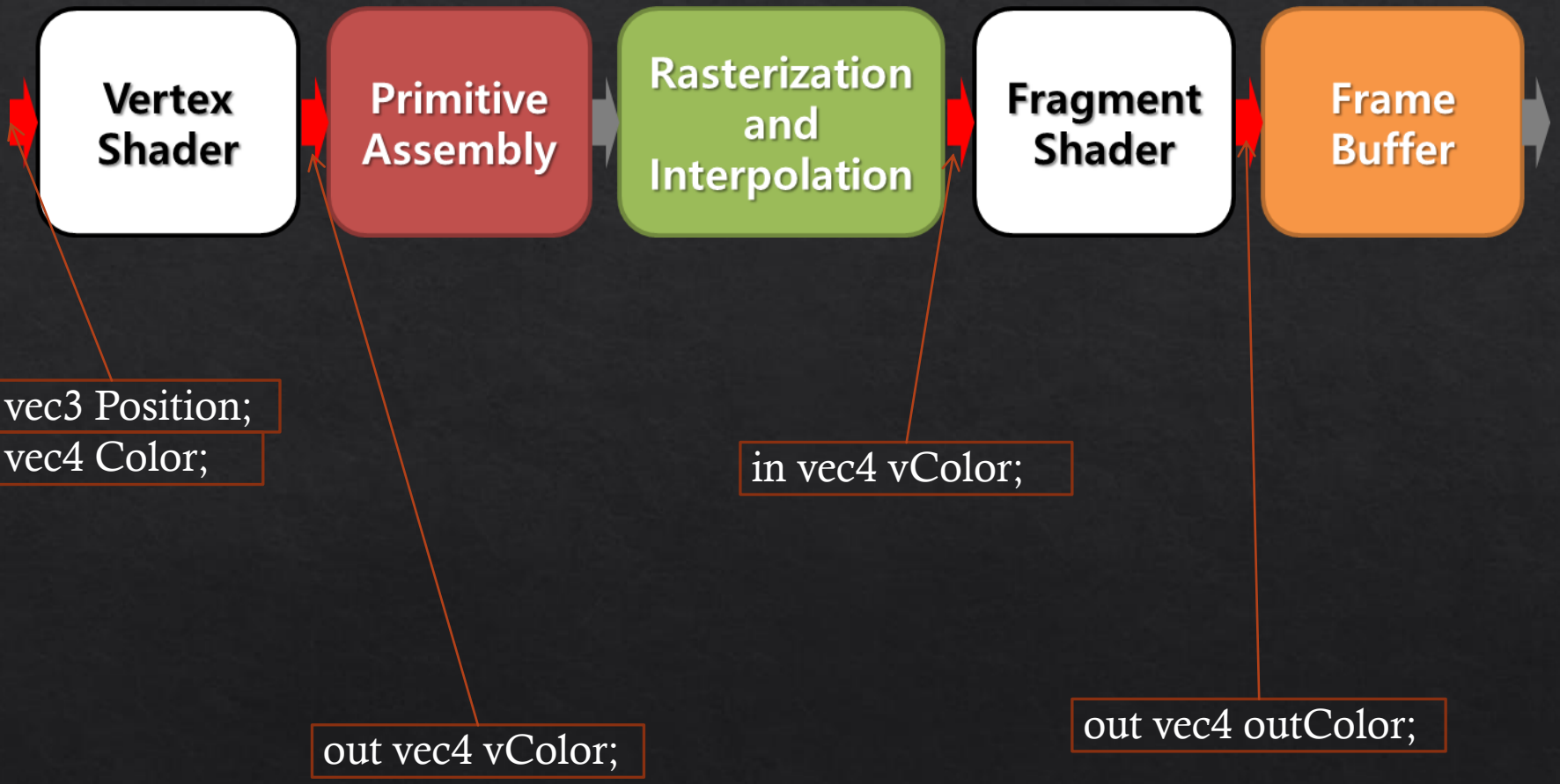
동일한 데이터 형태와 동일한 데이터 이름으로 주고 받아야 한다.
vs와 fs를 컴파일 하는 과정에서 링킹이 일어난다. 링크 과정에서
변수,함수 검사하는데 이름으로 검사를 한다. 세트가 맞아야만 엮이게 된다.
반드시 형태와 이름이 같아야 한다.

Storage Qualifier(in/out)

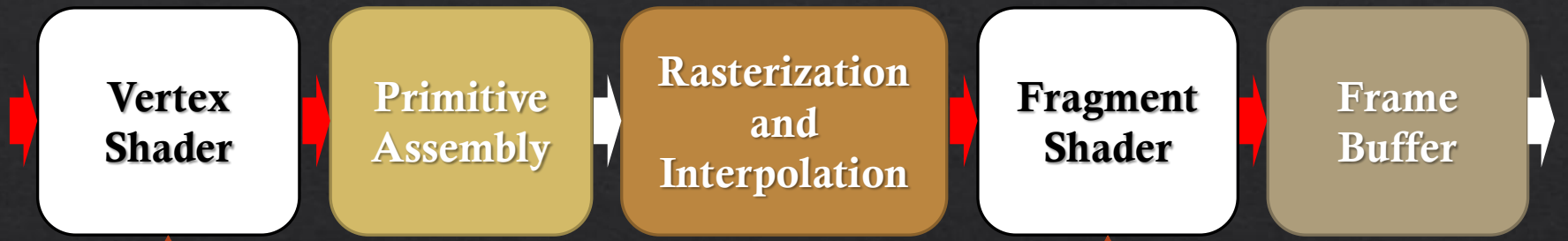




Storage Qualifier(in/out)



Storage Qualifier(uniform)



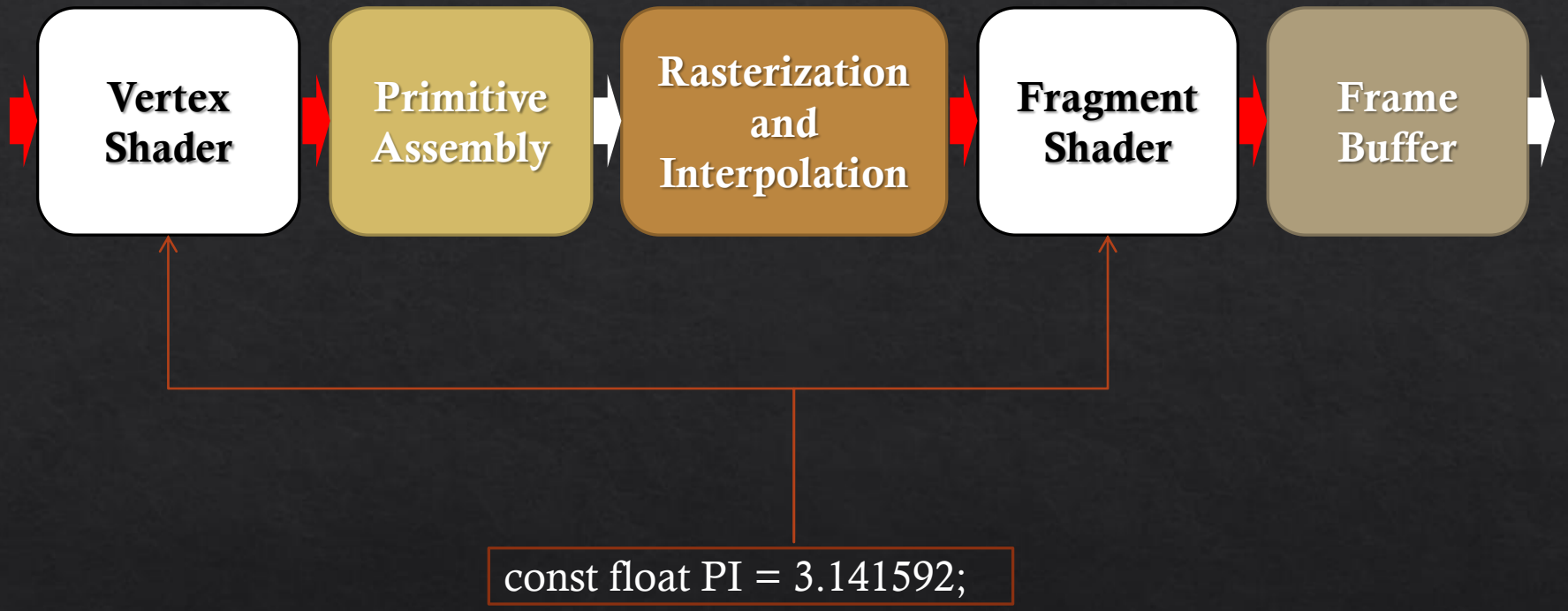
uniform float Time;
uniform mat4 MVPMatrix;

uniform float Time;
uniform float ColorLevel;

외부에서 지정한 값을 쓰고 싶을 경우 uniform 사용

uniform float Time; 과 같이 stage 간에 공유 가능함

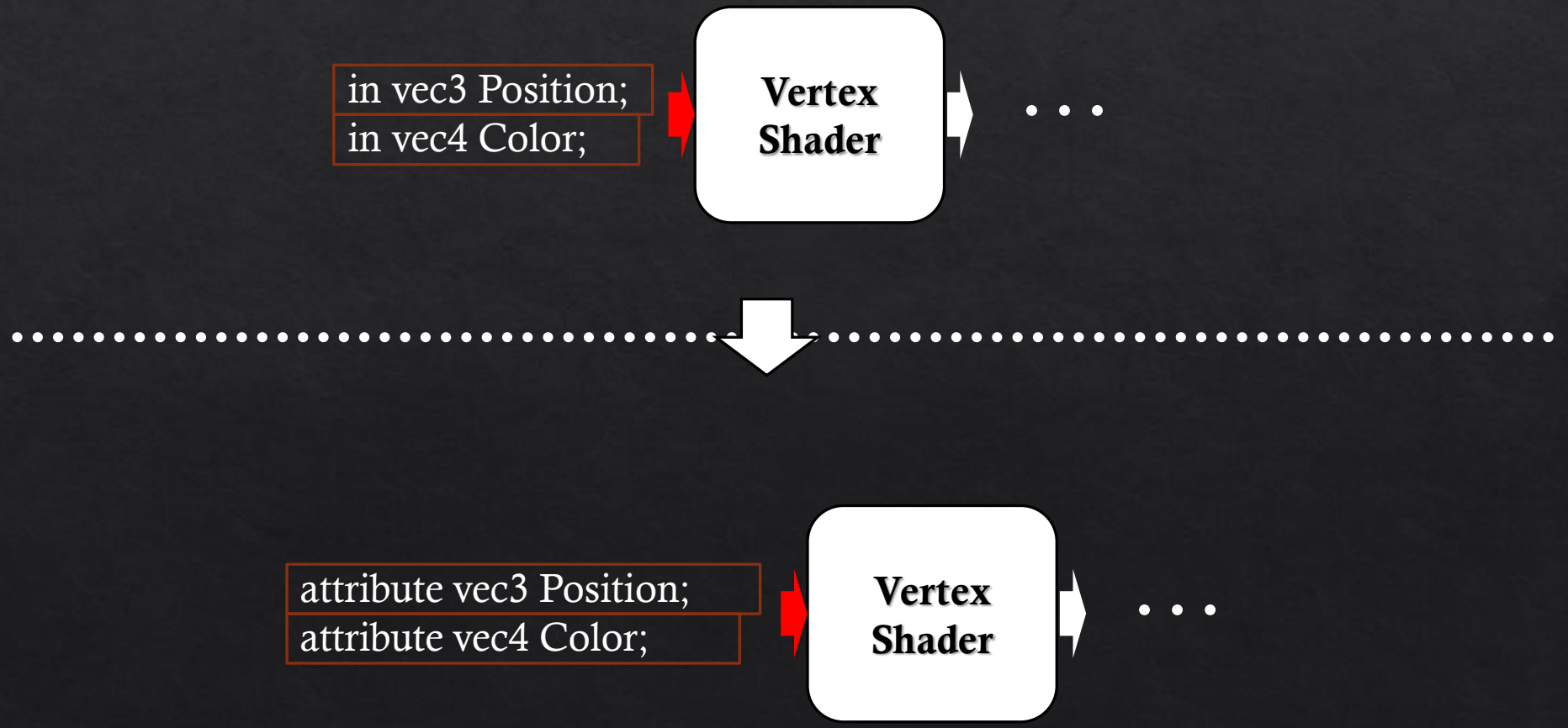
Storage Qualifier(const)



내부에서 지정하고 변하지 않을 경우 const 사용
각 셰이더 별로 선언 해야 함

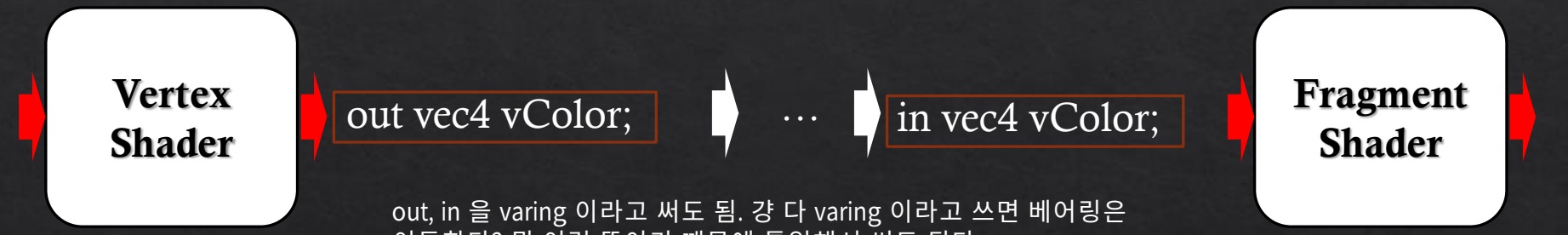
Storage Qualifier 사용

- 버텍스 셰이더의 경우 입력값 Qualifier 를 Attribute 바꾸어도 무방

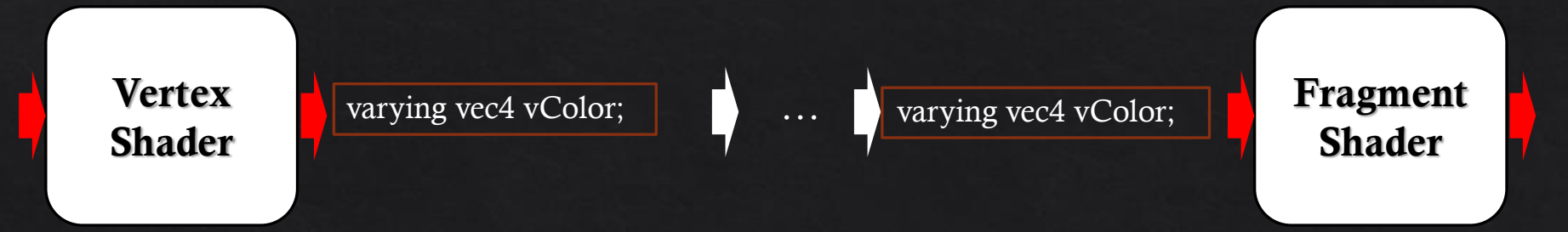


Storage Qualifier 사용

- 버텍스 셰이더의 out 과 프래그먼트 쉐이더의 in 의 경우 varying 으로 바꾸어도 무방



out, in 을 varing 이라고 써도 됨. 걍 다 varing 이라고 쓰면 베어링은 이동한다? 뭐 이런 뜻이기 때문에 통일해서 써도 된다.



무조건 시험에 나옴. 시험에 내기 좋음.

버텍스 셰이더 입력 데이터 패킹

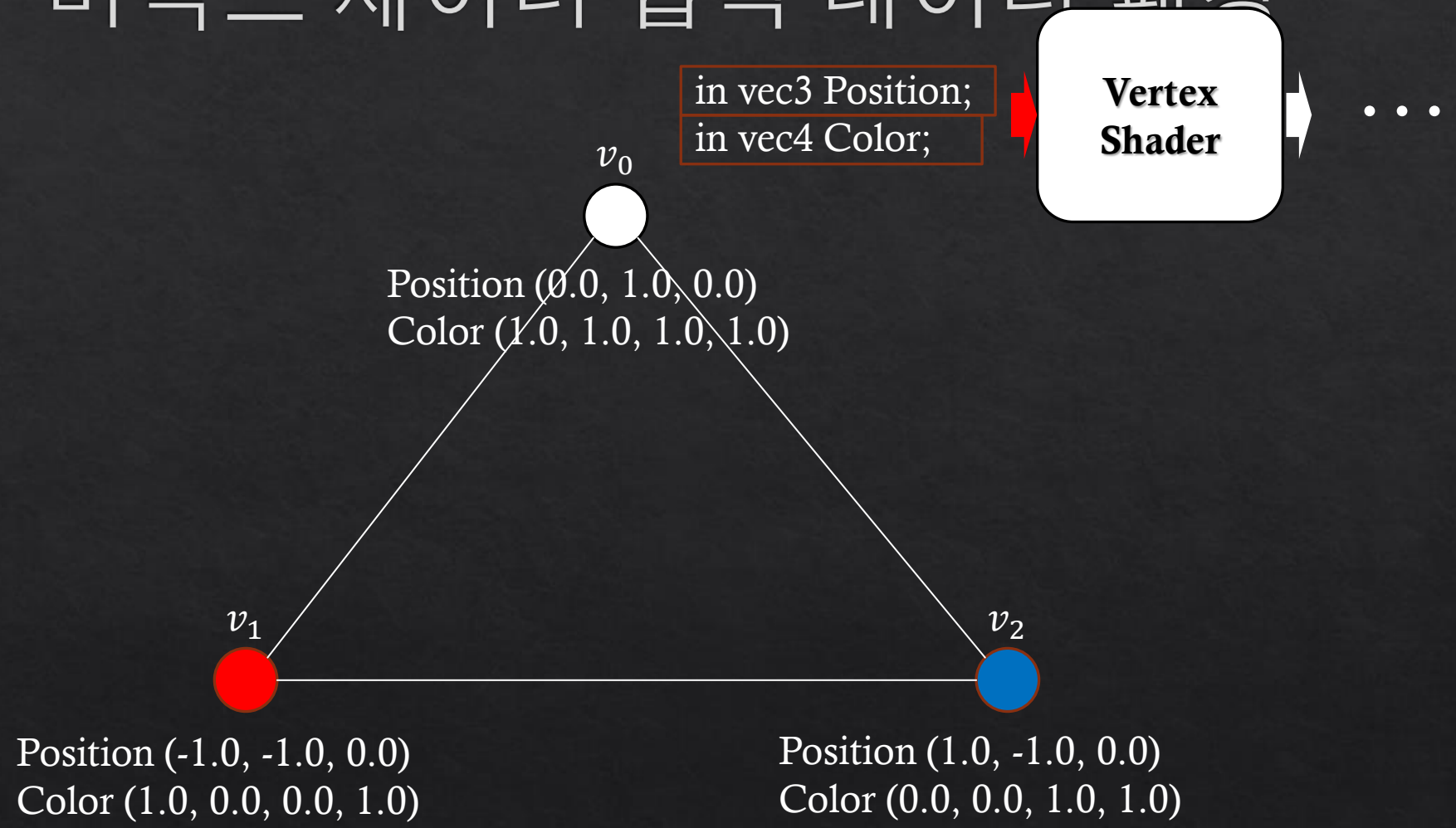
셰이더 프로그램 안에 attribute를 생성하고, enable 한 뒤 바인드하고 포인터 주는 형식의 코드를 썼다.
근데 이제 다른 여러 가지 방법을 소개해주겠다.

버텍스 셰이더 입력 데이터 패킹


```
int positionAttribID glGetAttribLocation(gShaderProgram, "Position");
int colorAttribID glGetAttribLocation(gShaderProgram, “Color");

glEnableVertexAttribArray(positionAttribID);
glEnableVertexAttribArray(colorAttribID);

glBindBuffer(GL_ARRAY_BUFFER, VBO); //?

glVertexAttribPointer(positionAttribID, ?, GL_FLOAT, GL_FALSE, ?, 0);
glVertexAttribPointer(positionAttribID, ?, GL_FLOAT, GL_FALSE, ?, 0);
```

버텍스는 세개밖에 없다고 치고, 컬러와 포지션이 있다고 가정.
데이터를 넘기는 세가지 방법이 있다. 뒷장에 나옴.

버텍스 셰이더 입력 데이터 패킹



버텍스 셰이더 입력 데이터 패킹

Position (0.0, 1.0, 0.0) Color (1.0, 1.0, 1.0, 1.0) v_0	Position (-1.0, -1.0, 0.0) Color (1.0, 0.0, 0.0, 1.0) v_1	Position (1.0, -1.0, 0.0) Color (0.0, 0.0, 1.0, 1.0) v_2

우리가 취하던 방법 = 1. 어트리뷰트가 추가될때마다 VBO가 추가되었음. 어트리뷰트와 VBO 의 개수가 같음. 근데 이러면 데이터 만드는 게 비효율적이고 길어짐. GPU 캐싱 문제도. VBO가 다르니까 둘이 완전히 다른 데이터라고 캐시가 인식함. 캐싱 사이즈가 9개라고 치면 VBO를 두개 따로 쓴다고 할 때 Position에서 3개, Color 에서 4 개 가져오면 이후에 쓸 때는 캐시를 완전히 지우고 다시 가지고 옴. 캐시에 다시 넣었다 뺐다가 계속 이루어짐. 캐시를 제대로 활용하지 못하게됨.

1. 각각 Array 생성

```
float Position[9] = {0.0, 1.0, 0.0, -1.0, -1.0, 0.0, 1.0, -1.0, 0.0};
float Color[12] = {1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0};
```

이렇게 합쳐서 올리는 게 효율적일 가능성 높음.

2. 합친 Array 생성 - 1

```
float PositionColor[21] = {0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 0.0, 1.0, 0.0, 0.0, 1.0,
1.0, -1.0, 0.0, 0.0, 0.0, 1.0, 1.0};
```

혹은 영역별로 나누어서 앞에는 포지션, 뒤에는 컬러 형식으로 만들 수 잇음. 그러나 이건 안 씀. 이것도 캐시에서 비효율적. 보통 1 아니면 2 쓴다.

3. 합친 Array 생성 - 2

```
float PositionColor[21] = {0.0, 1.0, 0.0, -1.0, -1.0, 0.0, 1.0, -1.0, 0.0, 1.0, 1.0, 1.0, 1.0,
1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0};
```

버텍스 셰이더 입력 데이터 패킹

뒷페이지에서 1 번의 장점은 컬러를 갈아치울 때 컬러 세트만 바꾸기 쉽다. 2 번의 경우는 컬러만 바꾸고 싶을 때 포지션 좌표들까지도 다 같이 조작해 줘야 함.
아무튼 보통은 2 번 형식으로 만든다. 이제 우리가 여태까지 1번 형식으로 만들던 것을 2번으로 바꿀 것이다. 포인터를 활용해서 몇개씩 띄워서 읽어들일지 이런 것 조작해 줘야 함.

1. 각각 Array 생성

```
float Position[9] = {0.0, 1.0, 0.0, -1.0, -1.0, 0.0, 1.0, -1.0, 0.0};
float Color[12] = {1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0};
```

버텍스 셰이더 입력 데이터 패킹


```
//OpenGL 데이터 생성
GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(Position), Position, GL_STATIC_DRAW);

GLuint VBO1;
glGenBuffers(1, &VBO1);
glBindBuffer(GL_ARRAY_BUFFER, VBO1);
glBufferData(GL_ARRAY_BUFFER, sizeof(Color), Color, GL_STATIC_DRAW);

//사용시
int positionAttribID glGetAttribLocation(gShaderProgram, "Position");
int colorAttribID glGetAttribLocation(gShaderProgram, “Color");

glEnableVertexAttribArray(positionAttribID);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glVertexAttribPointer(positionAttribID, 3, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(colorAttribID);
glBindBuffer(GL_ARRAY_BUFFER, VBO1);
glVertexAttribPointer(colorAttribID, 4, GL_FLOAT, GL_FALSE, 0, 0);

glDrawArrays(GL_TRIANGLES, 0, 3);
```

버텍스 셰이더 입력 데이터 패킹

2. 합친 Array 생성 - 1

```
float PositionColor[21] = {0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 0.0, 1.0,
0.0, 0.0, 1.0, 1.0, -1.0, 0.0, 0.0, 0.0, 1.0, 1.0};
```

버텍스 셰이더 입력 데이터 패킹


```
//OpenGL 데이터 생성
GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(PositionColor), PositionColor, GL_STATIC_DRAW);

//사용시
int positionAttribID glGetAttribLocation(gShaderProgram, "Position");
int colorAttribID glGetAttribLocation(gShaderProgram, “Color");

glEnableVertexAttribArray(positionAttribID);
glEnableVertexAttribArray(colorAttribID);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glVertexAttribPointer(positionAttribID, 3, GL_FLOAT, GL_FALSE, 7*sizeof(float), 0);
glVertexAttribPointer(colorAttribID, 4, GL_FLOAT, GL_FALSE, 7*sizeof(float), (GLvoid*)(3*sizeof(float)));

glDrawArrays(GL_TRIANGLES, 0, 3);
```

요 빨간 부분이 다른 부분. 스트라이드라는 개념이 이제 나온다.
현재 x, y, z, r, g, b, a가 나옴. 이 다음에도 이와 같은 형식이 반복해서 나올 것. x, y, z까지는 바로 읽을 수 있는데 이 다음부터는 포인트 몇 개를 띄어넘겨야 rgba가 나올지 알려줘야 한다. x0이 0번째로 나온다면 x1은 7번째로 나온다. 7 만큼의 어드레스의 차이가 생긴다. float이 7개니까 sizeof(float) * 7이다. 이것을 스트라이드라고 한다. 포지션에 관련된 정보는 0번째 이후 7번째 넘기고 다시 등장한다. 다음은 또 14번째에 등장....
즉, 시작지점과 몇개가 필요한지, 다음 시작은 어디인지를 표현하면 된다.

컬러 같은 경우는 0번째부터 읽으면 안 됨. 시작점이 0이 아니다. 3번을 넘어가고 나서 컬러값이 시작한다.
이후에는 7개를 넘기고 나서 다시 컬러값이 나올 것. 따라서 스트라이드는 포지션과 같은데 시작점만 다르다.
그래서 시작점을 저렇게 3*sizeof(float)으로 쓴 거.
시작지점은 (GLvoid*)로 타입캐스팅을 해서 적어줘야 한다.

버텍스 셰이더 입력 데이터 패킹

2. 합친 Array 생성 - 2

```
float PositionColor[21] = {0.0, 1.0, 0.0, -1.0, -1.0, 0.0, 1.0, -1.0, 0.0, 1.0, 1.0,
1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0};
```

버텍스 셰이더 입력 데이터 패킹


```
//OpenGL 데이터 생성
GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(PositionColor), PositionColor, GL_STATIC_DRAW);

//사용시
int positionAttribID glGetAttribLocation(gShaderProgram, "Position");
int colorAttribID glGetAttribLocation(gShaderProgram, “Color");

glEnableVertexAttribArray(positionAttribID);
glEnableVertexAttribArray(colorAttribID);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glVertexAttribPointer(positionAttribID, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(colorAttribID, 4, GL_FLOAT, GL_FALSE, 0, (GLvoid*)(9*sizeof(float)));

glDrawArrays(GL_TRIANGLES, 0, 3);
```

코드 상에서는 모든 버텍스들의 포지션이 똑같다. 그렇지만 버텍스 쉐이더에 들어간 뒤에 각자 포지션이 바뀌게 됨.
화면에 어느 위치에 그릴 것이냐 결정하는 것이 버텍스 쉐이더가 하는 역할이다. 이것이 가장 중요한 역할.

프래그먼트 쉐이더로 옮겨간 뒤에는 위치를 바꿀 수는 없음. 그러나 컬러를 변경할 수 있다. 위치는 VS, 컬러는 FS 담당.

실습

실습

- 파티클 별 랜덤 컬러 부여

- 파티클이 서서히 사라지도록

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

자료에 추가된 부분

실습

- 화면에 커다란 사각형 그리기 (색은 흰색으로)
  - Min(-0.5, -0.5), Max(0.5, 0.5)

- Varying 값을 사용하여 Interpolation 과정 이해
  - 버텍스 셰이더 출력 값 변화시켜 보기
  - 내부가 채워진 원 그려보기
  - 내부가 빈 원 그려보기
  - 여러 개의 동심원 그려보기
  - 특정 지점에 원 그려보기
  - 레이더 구현해 보기