Advanced OpenGL (1)

GPU Programming

2022학년도
2학기

ⓒ nvidia

Depth Testing

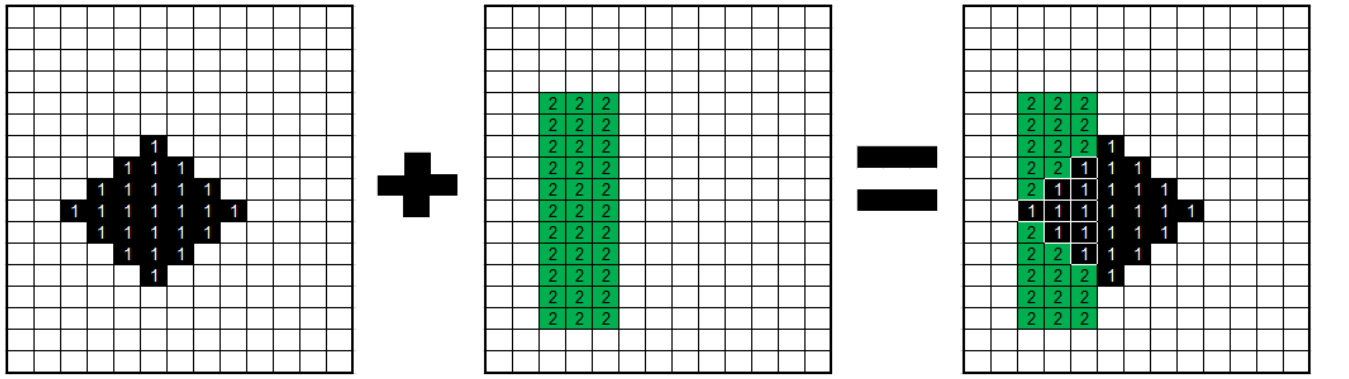
Depth Testing

- 깊이 버퍼 (depth buffer, aka Z-buffer)
  - 컬러 버퍼(모든 fragment의 최종 출력 색상을 저장하는 버퍼)와 마찬가지로 버퍼의 한 종류
  - Fragment의 깊이 정보를 저장하고, 일반적으로 color buffer와 동일한 해상도로 생성
  - 16, 24, 32비트 실수형(fixed-point 또는 floating-point)으로 저장되며, 대부분의 시스템은 24비트 사용
  - 지금까지 예제 코드에서 따로 depth buffer를 만들지 않았던 이유는? GLFW가 자동 생성해주었기 때문
- 깊이 검사(depth testing)
  - OpenGL이 깊이 버퍼의 내용에 따라 fragment의 깊이 값을 검사
  - 이 검사를 통과하면 깊이 버퍼의 해당 픽셀 값은 새로운 깊이 값으로 갱신
  - 이 검사를 실패하면 해당 fragment는 폐기
  - Fragment shader (및 스텐실 검사가) 수행된 후, gl_FragCoord.z 값을 이용해 screen space에서 수행
- OpenGL에서의 깊이 검사 사용법
  - 깊이 검사를 활성화시키기 위해서는 glEnable(GL_DEPTH_TEST) 필요 `glEnable(GL_DEPTH_TEST);`
  - 매 프레임마다 컬러 버퍼와 깊이 버퍼를 함께 clear `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`
  - 만약 깊이 버퍼를 갱신하고 싶지 않으면 아래와 깊이 마스크를 사용하지 않도록 수정

```
glDepthMask(GL_FALSE);
```

Early Z-test

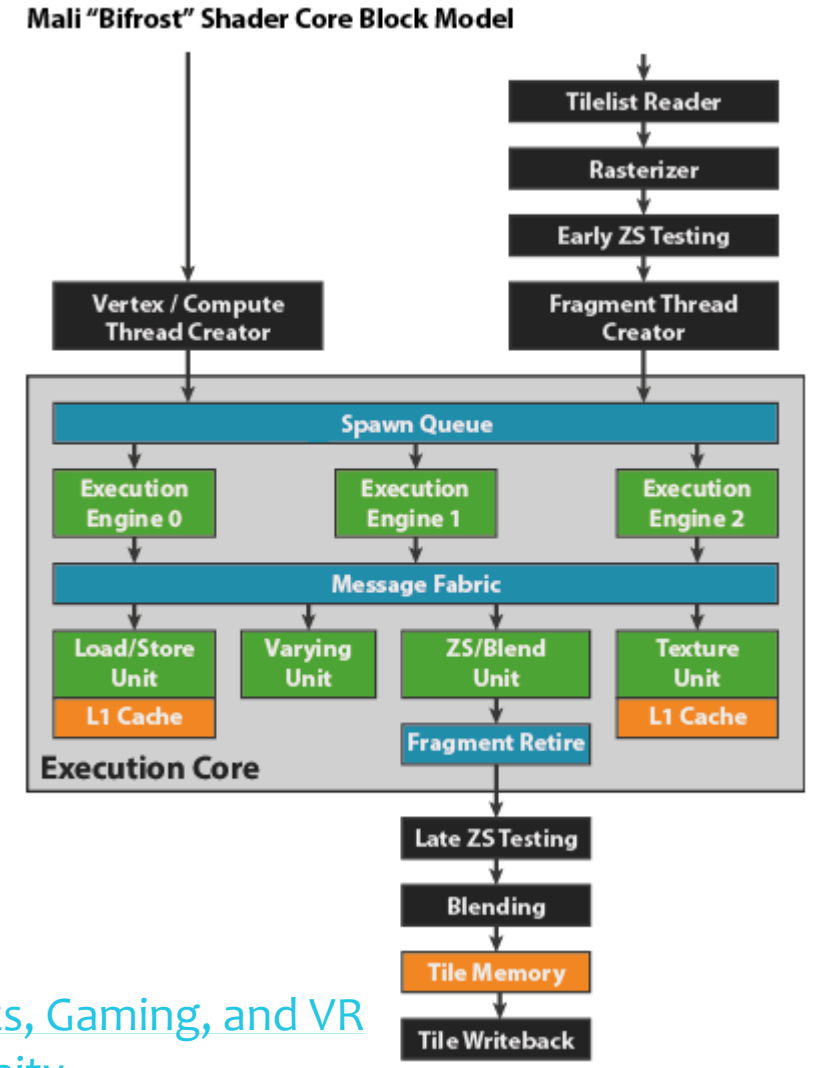
- Early Z-test
  - GPU H/W 상에서 필요시 fragment shading 전에 미리 깊이 검사 수행
  - 보여지지 않을 fragment는 값비싼 shader 연산 전에 미리 폐기하고, 깊이 검사를 통과하면 깊이 버퍼의 값을 shading 전에 미리 수정
  - 프로그램 상에서 별도 명령을 내리지 않아도, 대부분의 GPU에서 자동으로 효율 증가를 위해 수행됨 (OpenGL 4.2 이상에서 이 기능 강제 가능)



Color buffer with existing rendered pixels with Z value 1

New pixels to render with Z value 2

Highlighted conflicting Pixels are rejected by Early Z rejection feature saving efforts to coloring conflicting pixels



Overview — Game Developer Guides documentation (qualcomm.com)

Mali GPU: The Bifrost Shader Core - Graphics, Gaming, and VR blog - Arm Community blogs - Arm Community

Early Z-test

- 자동 early z-test 수행시 몇 가지 제약 사항
  - Fragment shader에서 깊이 값을 직접 수정하지 않아야 함
  - Fragment가 FS 상에서 discard 명령으로 폐기되지 않아야 함
  - 그 이유는 early z-test의 결과로 갱신한 결과가 invalid해지기 때문

- 단, 위 제약 사항은 OpenGL 4.2 이상에서 early z-test를 강제하면 무시됨
  - 이를 강제하기 위해서는 FS상에 아래 행 추가

    layout(early_fragment_tests) in;

Early Z-test

- 효과적인 early-z test로 visible fragment를 잘 선별하는 것은 GPU의 성능과 직결됨
  - 각 GPU vendor별로 여러가지 다른 H/W 구조 사용
  - 예를 들면, z-buffer를 밉맵처럼 계층적으로 구성 가능 greene93.pdf (princeton.edu)


300

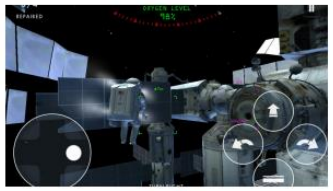

Captain America


Sniper 3D Assassin

Hot Wheels: Race Off


Beach Buggy Racing


Counter Strike


Gravity


Temple Run

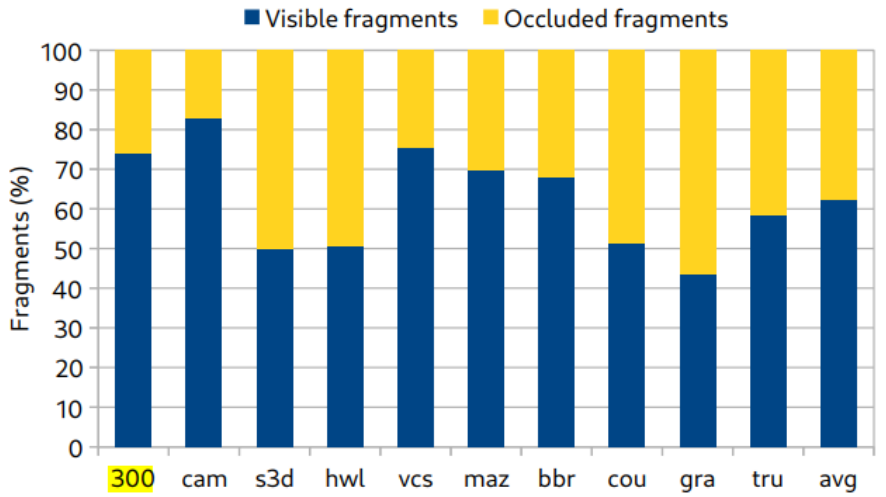


Fig. 2. Processed fragments broken down into visible and occluded ones for the evaluated benchmarks. Occluded fragments represent the overall amount of overdraw, and hence, a waste of resources.

Omega_Test__TVCG_journal____R3___FINAL_version.pdf (upc.edu)

Depth Test Function

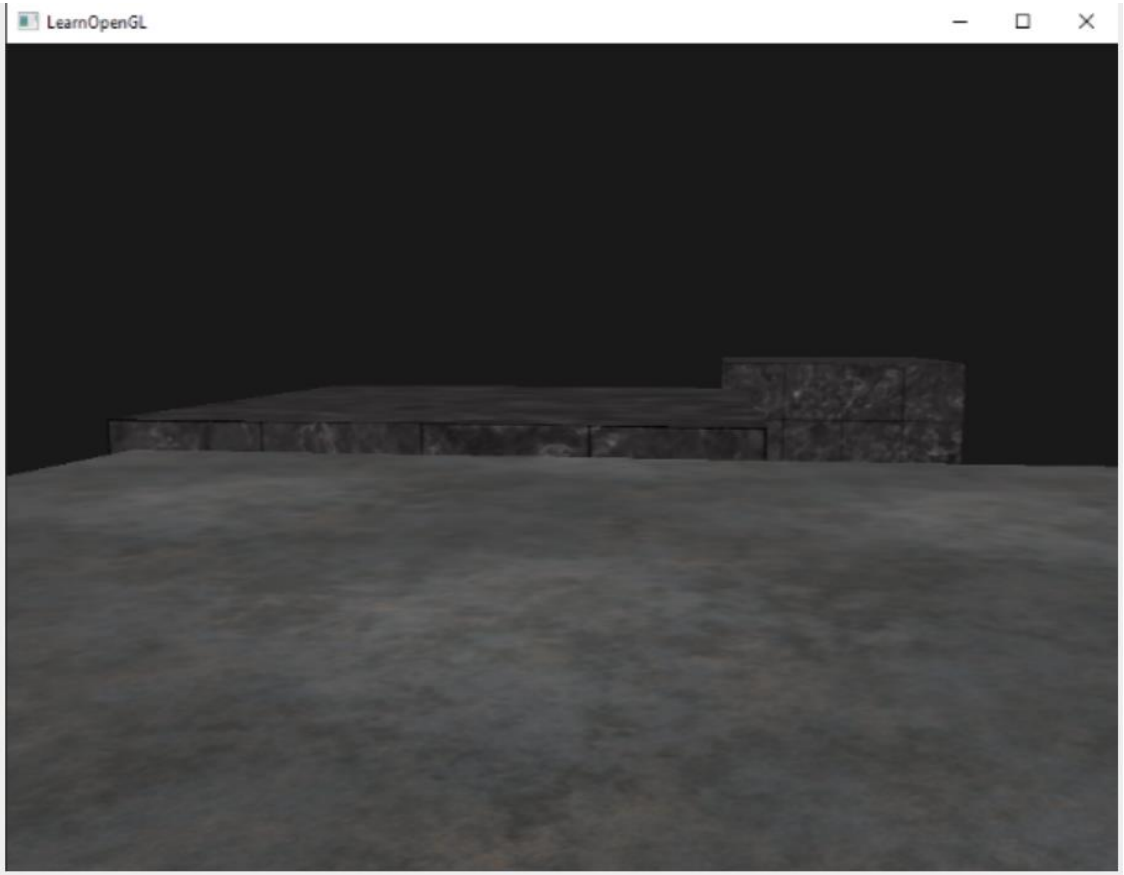
- OpenGL은 깊이 검사의 동작 방식을 지정 가능 `glDepthFunc(GL_LESS);`

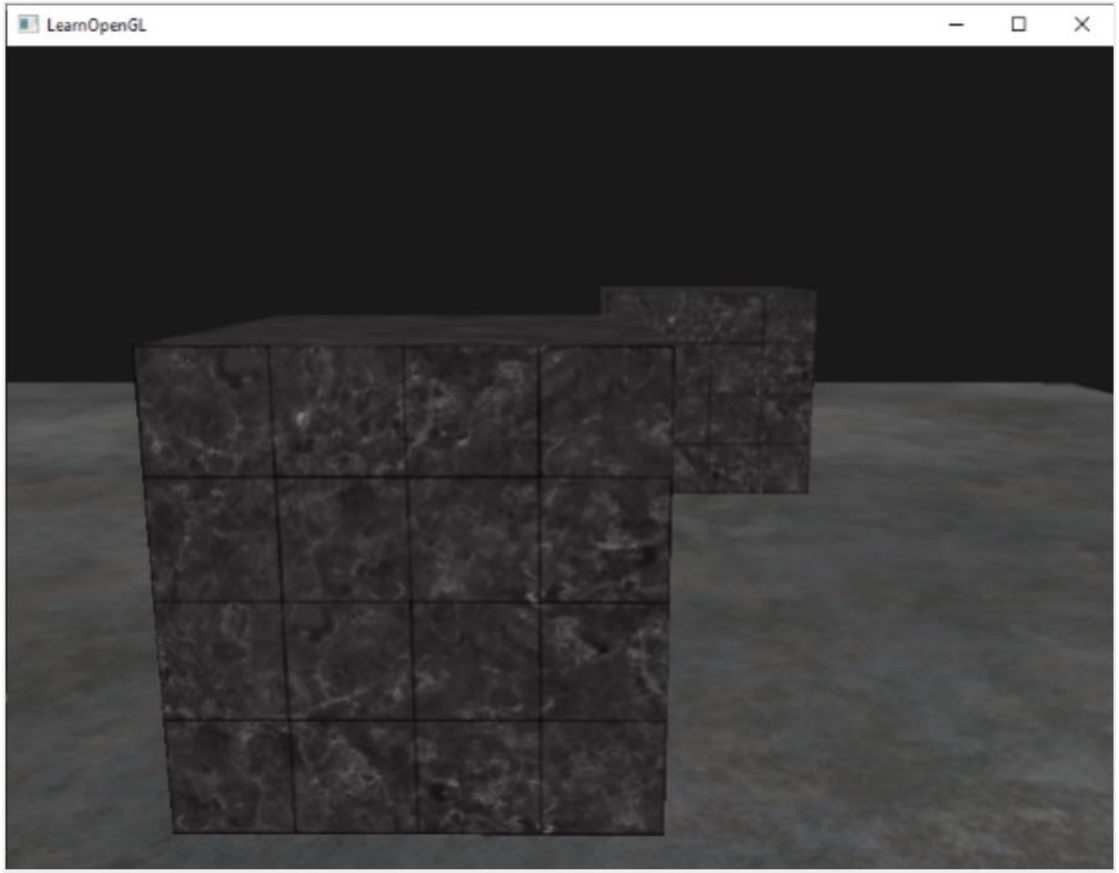
Function	Description
GL_ALWAYS	The depth test always passes.
GL_NEVER	The depth test never passes.
GL_LESS	Passes if the fragment's depth value is less than the stored depth value.
GL_EQUAL	Passes if the fragment's depth value is equal to the stored depth value.
GL_LEQUAL	Passes if the fragment's depth value is less than or equal to the stored depth value.
GL_GREATER	Passes if the fragment's depth value is greater than the stored depth value.
GL_NOTEQUAL	Passes if the fragment's depth value is not equal to the stored depth value.
GL_GEQUAL	Passes if the fragment's depth value is greater than or equal to the stored depth value.

기본값

Depth Test Function

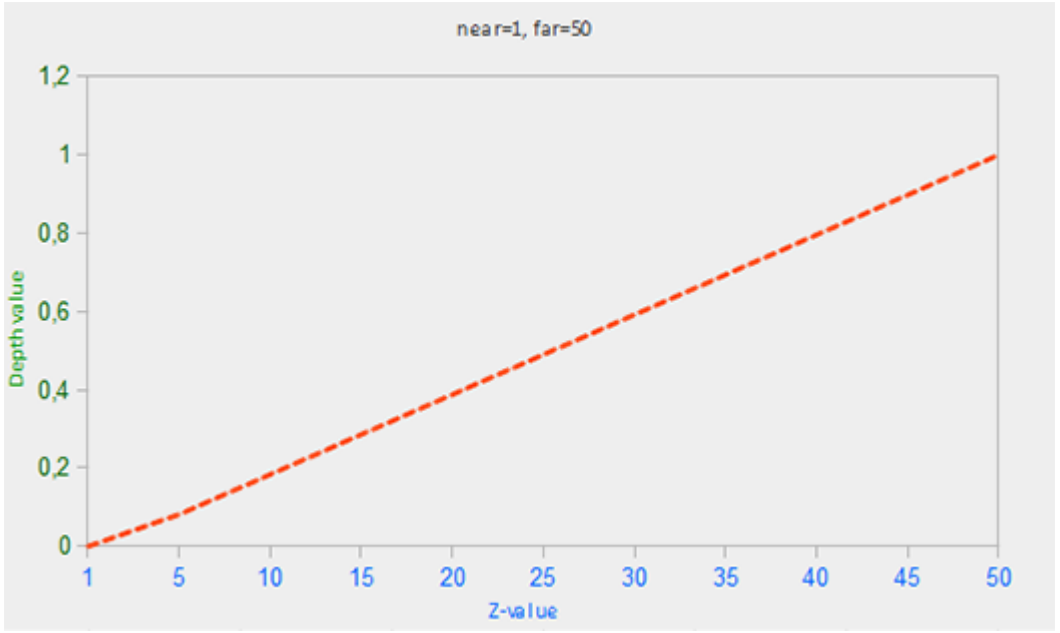
- 컨테이너가 앞에 있는 장면에서, 깊이 검사 함수 설정에 따른 렌더링 결과 비교
  - GL_ALWAYS 사용시
  - GL_LESS 사용시

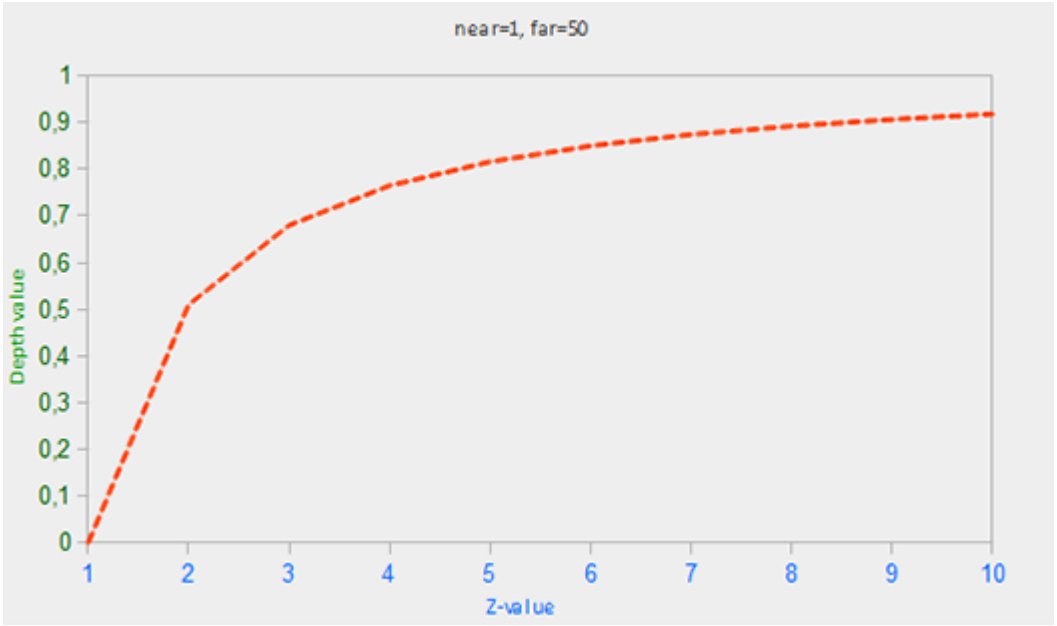




Depth Value Precision

- 깊이 버퍼에는 view-space의 z값이 [0, 1] 범위로 변환된 depth값이 들어가야 함
  - 선형(왼쪽)보다는 비선형(오른쪽) 계산식이 주로 사용됨.
    그 이유는 가까이 있는 물체에 더 높은 정밀도를 부여하기 위함



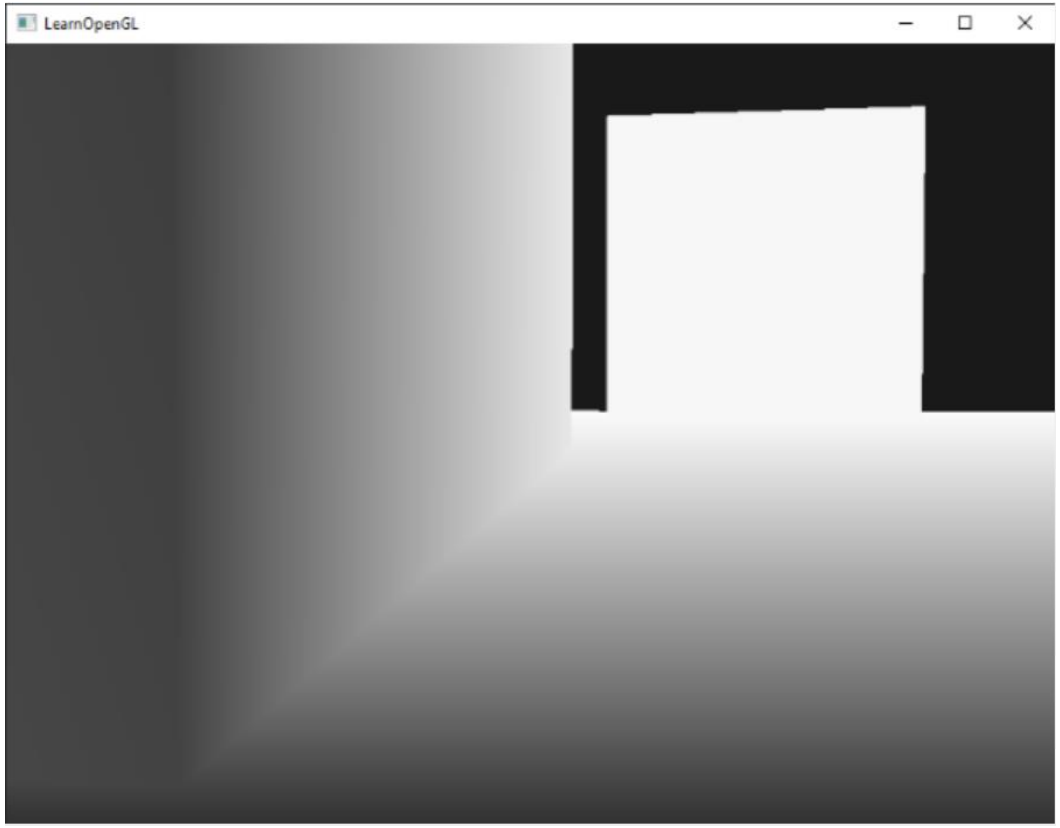


$$F_{depth} = \frac{z - near}{far - near}$$

$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$

Visualizing the depth buffer

- 실제 깊이 버퍼에 저장된 결과 (비선형)



FS

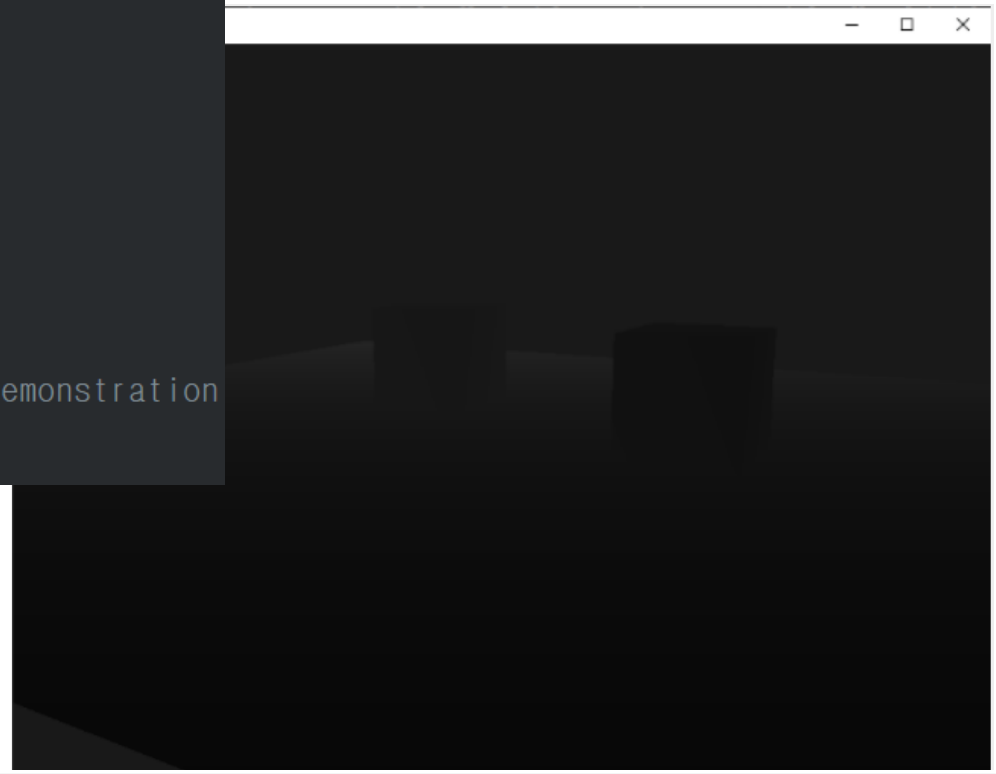
```
void main()
{
    FragColor = vec4(vec3(gl_FragCoord.z), 1.0);
}
```

Visualizing the depth buffer

- 강제로 선형으로 만든 깊이 버퍼
  - near plane: 0.1, far plane: 100

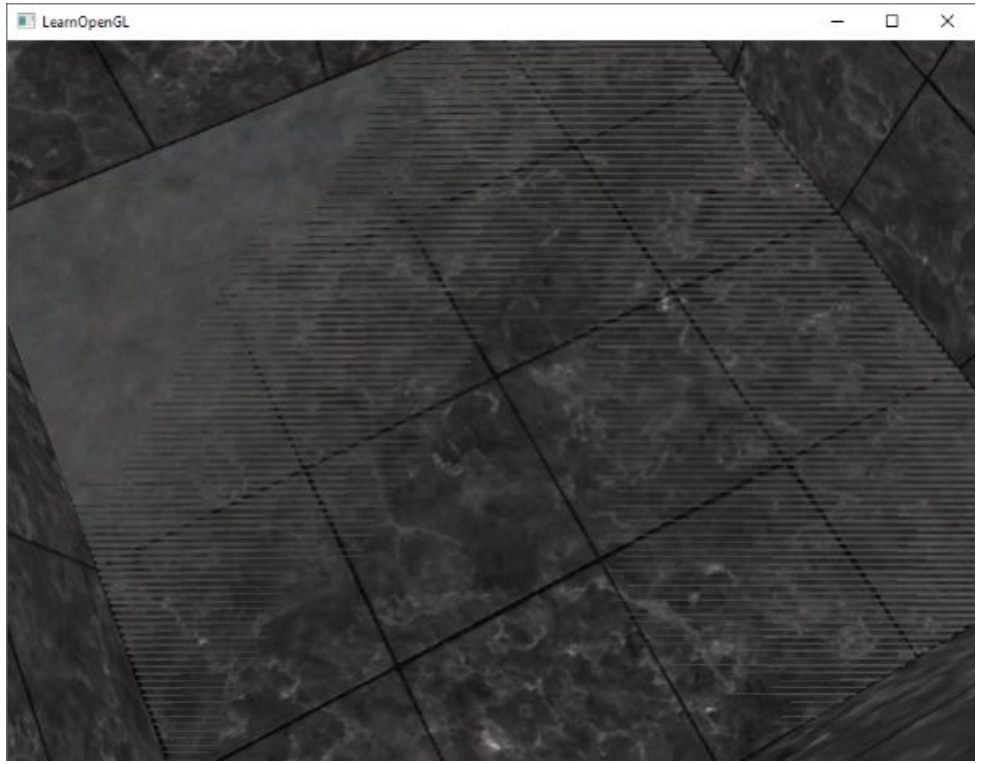
FS

```
#version 330 core
out vec4 FragColor;

float near = 0.1;
float far  = 100.0;

float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // back to NDC
    return (2.0 * near * far) / (far + near - z * (far - near));
}

void main()
{
    float depth = LinearizeDepth(gl_FragCoord.z) / far; // divide by far for demonstration
    FragColor = vec4(vec3(depth), 1.0);
}
```



Z-fighting

- 두 평면이나 삼각형이 아주 가깝게 서로 나란히 위치할 때 생길 수 있는 시각적 결함 (artifact)
  - 깊이 버퍼가 충분한 정밀도를 가지지 못할 경우 두 평면 또는 삼각형의 순서가 계속해서 바뀌는 것처럼 보임
  - 둘 중 누가 앞에 있는지 판단하지 못하고 싸우는 것처럼 보이므로 z-fighting이라 불림
- 정밀도 문제이므로 완벽하게 해결할 수는 없음



Prevent Z-fighting

- 삼각형이 겹쳐지지 않도록 아주 작은 offset을 생성
  - 각 물체별로 수작업 필요

- Near 평면을 가능한 한 시점으로부터 멀리 설정
  - Near 평면에 가까운 물체들의 깊이 버퍼 정밀도를 높임
  - 단, 가까이 있는 물체들을 clipping할 수 있으므로 주의 (실험을 통해 최적의 near 거리를 설정해야 함)

- 높은 정밀도의 깊이 버퍼 사용
  - 16비트(fixed-point) 대신 24비트(fixed-point)를, 24비트(fixed-point) 대신 32비트 floating-point를 사용
  - 메모리 사용량 증가 및 성능 감소 가능성

ⓒ nvidia

Stencil Testing

Stencil Testing

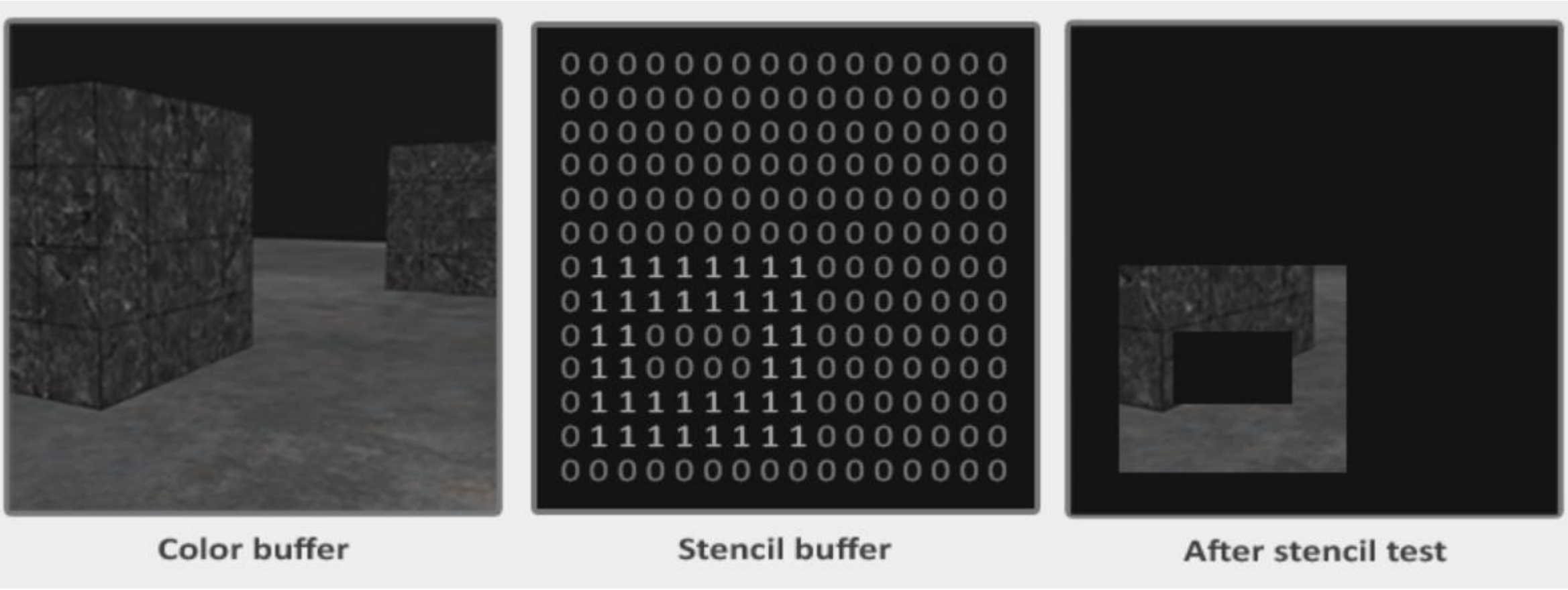
- 스텐실 버퍼(stencil buffer)를 기반으로 깊이 검사 전에 수행되는, fragment를 폐기할지 말지 검사하는 또다른 기준
  - 미술에서의 스텐실 기법과 유사

- 스텐실 버퍼
  - 일반적으로 픽셀당 8비트의 스텐실 값(0~255)을 가짐
  - 스텐실 버퍼 역시 GLFW에서 자동으로 생성 (생성 여부는 라이브러리에 따라 다름)

아동미술 / 스텐실 기법으로 꾸미기 - YouTube

Stencil Testing

- 스텐실 검사의 예시



Stencil Testing

- 스텐실 검사의 순서
  - 스텐실 버퍼의 writing을 활성화 (스텐실 마스크 설정)
  - 물체를 그린 후 스텐실 버퍼 수정
  - 스텐실 버퍼의 writing을 비활성화 (스텐실 마스크 설정)
  - 스텐실 버퍼를 기반으로 특정 fragment를 폐기하여 물체를 렌더링

- OpenGL에서의 스텐실 검사 방법
  - 깊이 검사를 활성화시키기 위해서는 glEnable(GL_STENCIL_TEST) 필요 `glEnable(GL_STENCIL_TEST);`
  - 매 프레임마다 스텐실 버퍼도 함께 clear

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

  - 스텐실 마스크 함수를 이용하여, 스텐실 값에 AND 연산을 시킬 bitmask를 설정

```
glStencilMask(0xFF); // each bit is written to the stencil buffer as is
glStencilMask(0x00); // each bit ends up as 0 in the stencil buffer (disabling writes)
```

Stencil Test & Operation Functions

- glStencilFunc(GLenum func, GLint ref, GLuint mask)
  - 스텐실 버퍼의 내용으로 스텐실 검사를 통해 fragment를 어떻게 처리할지 결정
  - func: 저장된 스텐실 값과 ref 값을 비교하는 스텐실 검사 함수 설정.
    (깊이 검사 함수와 유사하게, 결과가 func와 같으면 pass)
    GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS
  - ref: 스텐실 검사에 대한 레퍼런스 값 지정
  - mask: 비교 전에 레퍼런스 값과 저장된 스텐실 값 모두에 AND 연산이 수행되어질 mask 지정

- glStencilFunc 사용 예시
  - Pass인 경우 – fragment의 stencil 값 == ref
  - Fail인 경우 – 그 외의 모든 결과

```
glStencilFunc(GL_EQUAL, 1, 0xFF)
```

Stencil Test & Operation Functions

- glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass)
  - 스텐실 버퍼를 어떻게 수정할지 설정
  - sfail: stencil test가 실패하였을 때 취할 행동 (기본값: GL_KEEP)
  - dpfail: stencil test가 통과했지만 depth test는 실패했을 때 취할 행동 (기본값: GL_KEEP)
  - dppass: stencil, depth test 모두 통과했을 때 취할 행동 (기본값: GL_KEEP)

Action	Description
GL_KEEP	The currently stored stencil value is kept.
GL_ZERO	The stencil value is set to 0.
GL_REPLACE	The stencil value is replaced with the reference value set with `glStencilFunc`.
GL_INCR	The stencil value is increased by 1 if it is lower than the maximum value.
GL_INCR_WRAP	Same as `GL_INCR`, but wraps it back to 0 as soon as the maximum value is exceeded.
GL_DECR	The stencil value is decreased by 1 if it is higher than the minimum value.
GL_DECR_WRAP	Same as `GL_DECR`, but wraps it to the maximum value if it ends up lower than 0.
GL_INVERT	Bitwise inverts the current stencil buffer value.

Object Outlining

- 각 물체별로 색이 입혀진 작은 외곽선을 생성하는 예시
  - 전략 게임에서 선택된 유닛을 표현할 때 사용 가능한 효과

- Outlining 과정
  - Stencil writing을 활성화
  - 물체를 그리기 전에 스텐실 함수를 GL_ALWAYS로 설정하고,
    물체의 fragment가 렌더링 될 때마다 스텐실 버퍼를 1로 수정하도록 설정
  - 물체를 렌더링
  - Stencil writing과 깊이 검사를 비활성화
  - 물체를 약간 확대
  - 물체의 외곽선을 출력하는 별도의 fragment shader 사용
  - 물체를 다시 그리지만, 스텐실 값이 1이 아닌 fragment들만 그림
  - 깊이 검사를 다시 활성화하고, 스텐실 함수를 GL_KEEP으로 복원

shaders - outline object effect - Game Development Stack Exchange

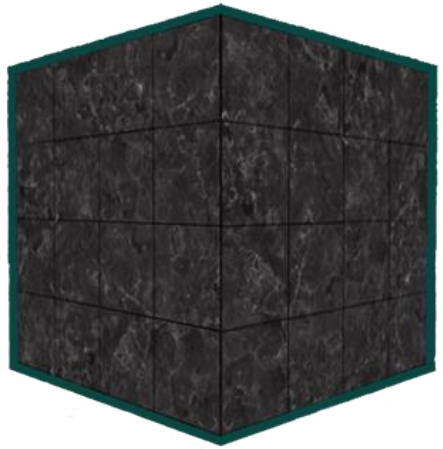
Object Outlining

- Outlining 구현
  - 외곽선 색상을 설정하는 셰이더 작성

```
void main()                                          FS
{
    FragColor = vec4(0.04, 0.28, 0.26, 1.0);
}
```

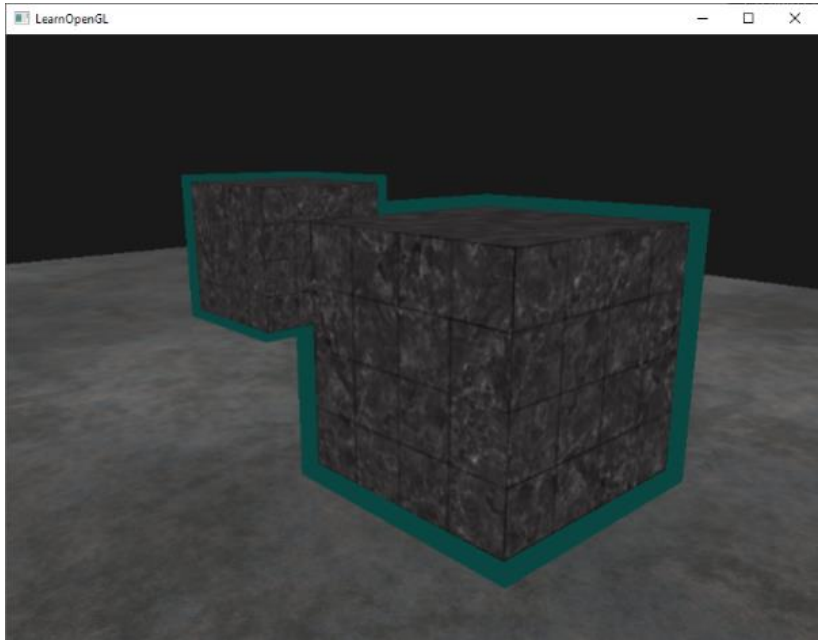
  - 스텐실 검사를 활성화한 후,
    바닥, 안쪽 물체, 외곽선을 차례대로 그림

```
glEnable(GL_DEPTH_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

glStencilMask(0x00); // make sure we don't update the stencil buffer while drawing the floor
normalShader.use();
DrawFloor()

glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilMask(0xFF);
DrawTwoContainers();

glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
glStencilMask(0x00);
glDisable(GL_DEPTH_TEST);
shaderSingleColor.use();
DrawTwoScaledUpContainers();
glStencilMask(0xFF);
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glEnable(GL_DEPTH_TEST);                             CPU
```



Object Outlining

- Outlining 구현 결과
  - 겹쳐지지 않는 외곽선을 원한다면, 물체마다 스텐실 버퍼를 비우고 깊이 버퍼를 적절히 생성
  - 자연스러운 외곽선을 원한다면, Gaussian blur와 같은 전처리 필터도 사용 가능



- 기타 응용 예
  - 백미러에 비치는 부분을 그릴 때
  - Shadow volume

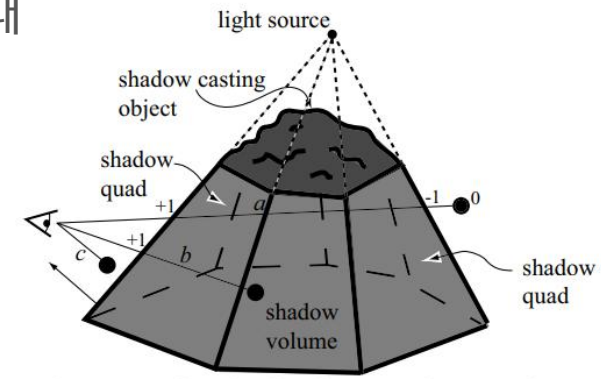


Figure 2: The standard shadow volume algorithm. The shadow volume here consists of seven quads. Ray b is inside shadow with a stencil value of 1. Ray a and c are outside shadow with stencil values of 0.

soft_hardware.dvi (chalmers.se)

ⓒ nvidia

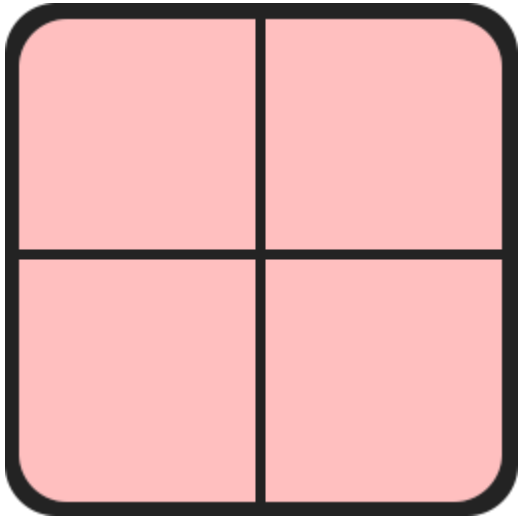
Blending

Blending

- 블렌딩은 투명한 물체를 구현하는 기술
  - 물체 자체의 색상과 뒤에 있는 다른 물체의 색상을 혼합(blend)
  - 혼합 정도는 보통 물체의 투명도(alpha값)에 의해 결정됨 – 0.0은 완전 투명, 1.0은 완전 불투명

Full transparent window

Partially transparent window



0.75의 alpha 값을 갖는 빨간 창문
blending_transparent_window.png (256×256) (learnopengl.com)

Discarding Fragments

- 완전 투명 또는 불투명한 영역만 가지는 잔디 텍스처의 예
  - Alpha값을 이용해 잔디 부분은 그리고, 투명한 영역은 그리지 않게 할 수 있음

- Alpha testing
  - 투명한 영역을 그리지 않기 위해서 fragment의 알파 값을 검사한 후 이를 통과하지 못한 fragment를 discard시키는 방법

- Alpha test를 이용한 잔디 렌더링 예
  - RGBA 텍스처를 GPU 메모리에 올림

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

CPU

  - FS에서도 텍셀의 a 채널값을 그대로 fragment에 적용

```
void main()
{
    // FragColor = vec4(vec3(texture(texture1, TexCoords)), 1.0);
    FragColor = texture(texture1, TexCoords);
}
```

FS

Discarding Fragments

- Alpha test를 이용한 잔디 렌더링 예 (cont.)
  - 사각형 잔디 객체들의 위치 설정

CPU

```
vector<glm::vec3> vegetation;
vegetation.push_back(glm::vec3(-1.5f,  0.0f, -0.48f));
vegetation.push_back(glm::vec3( 1.5f,  0.0f,  0.51f));
vegetation.push_back(glm::vec3( 0.0f,  0.0f,  0.7f));
vegetation.push_back(glm::vec3(-0.3f,  0.0f, -2.3f));
vegetation.push_back(glm::vec3( 0.5f,  0.0f, -0.6f));
```

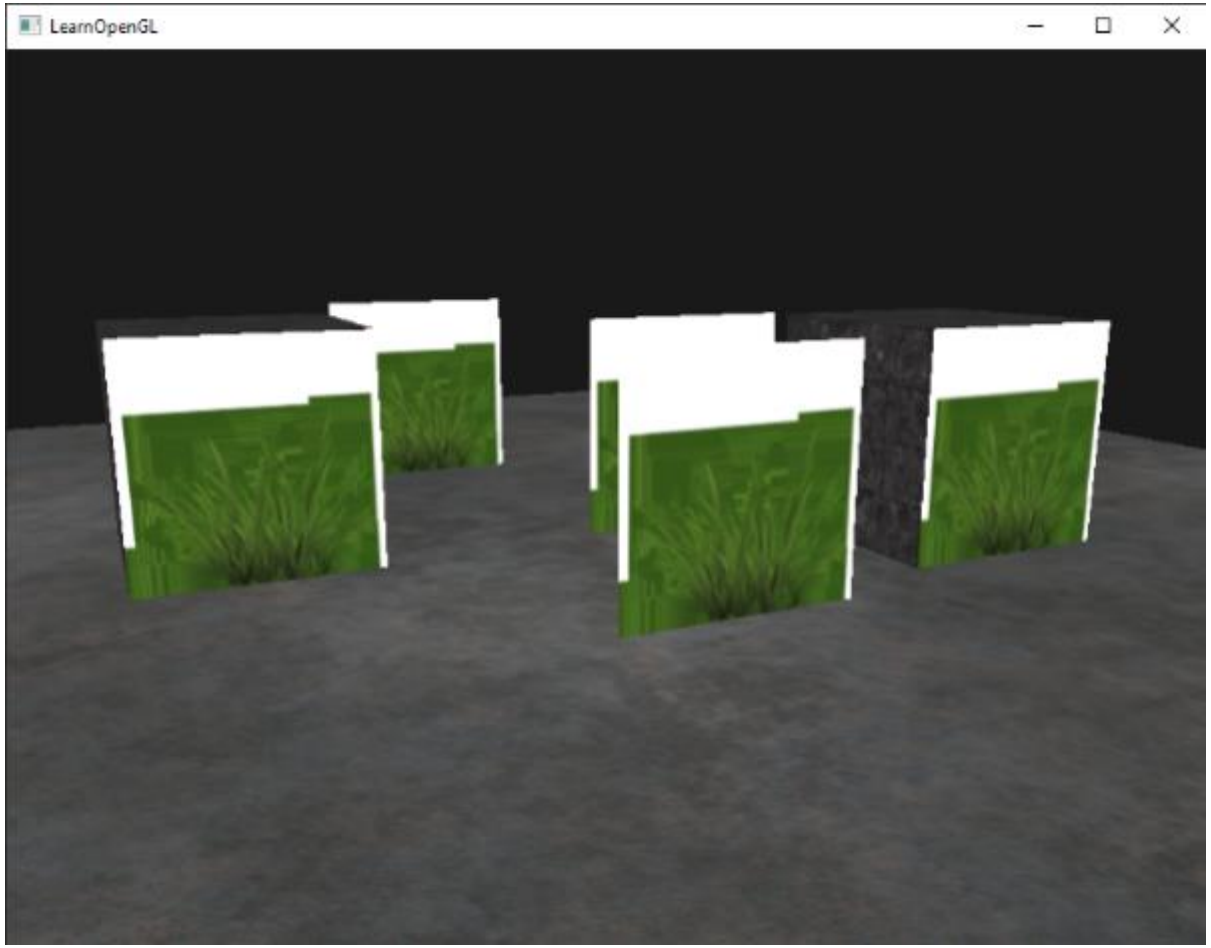
  - 잔디 텍스처를 입힌 각 사각형 객체를 렌더링

CPU

```
glBindVertexArray(vegetationVAO);
glBindTexture(GL_TEXTURE_2D, grassTexture);
for(unsigned int i = 0; i < vegetation.size(); i++)
{
    model = glm::mat4(1.0f);
    model = glm::translate(model, vegetation[i]);
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Discarding Fragments

- Alpha test를 이용한 잔디 렌더링 예 (cont.)
  - 중간 실행 결과 (2개의 컨테이너 + 5개의 잔디 사각형)



Discarding Fragments

- Alpha test를 이용한 잔디 렌더링 예 (cont.)
  - 텍스처의 alpha값에 따라 fragment를 그릴지 말지 결정하도록 FS 수정
  - 아래 예시에서는 alpha값 0.1 미만의 투명한 영역은 그리지 않게 됨

FS

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    FragColor = texColor;
}
```

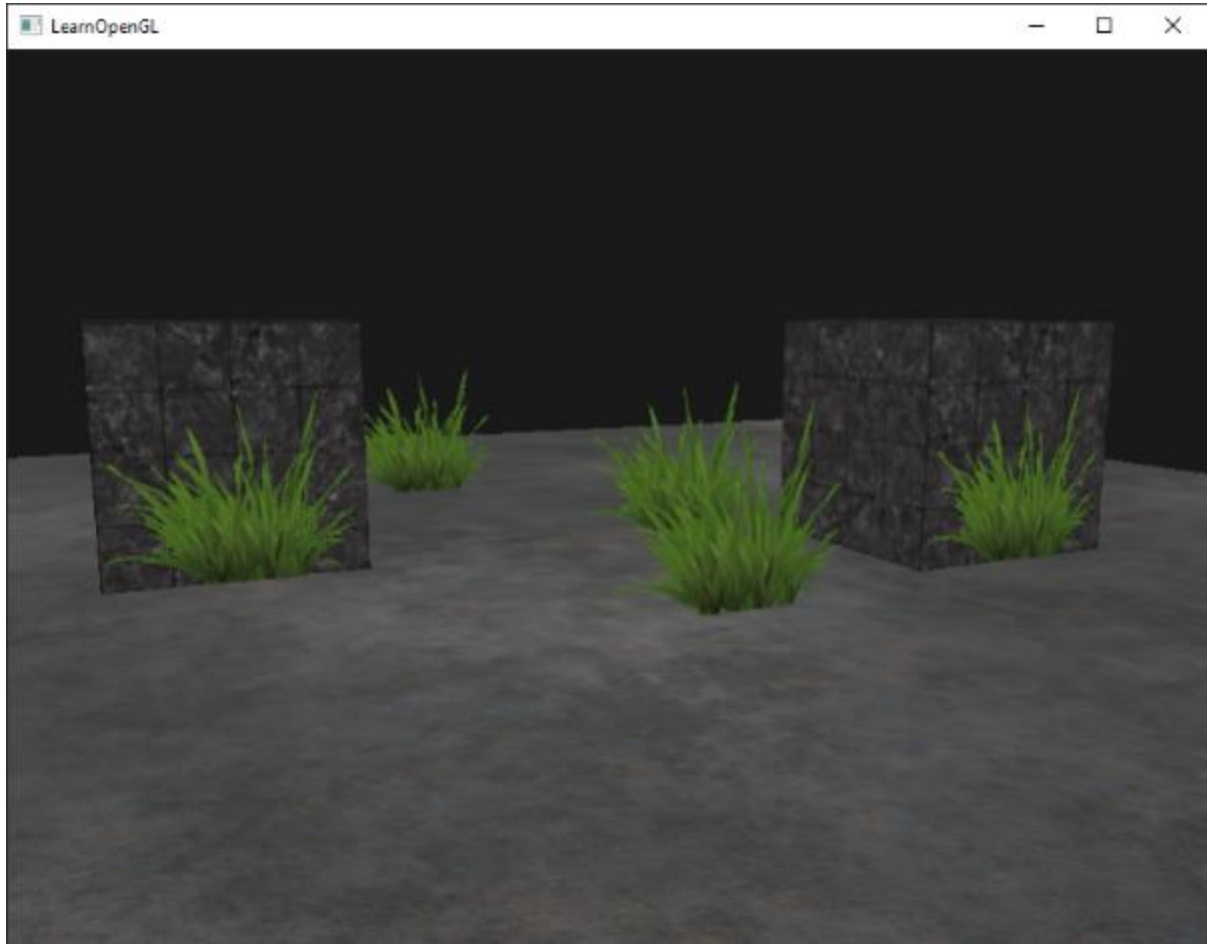
  - GL_REPEAT로 wrapping시 텍스처 사각형에 반투명한 경계가 나타날 수 있으므로, 텍스처 사각형 렌더링시 이 옵션을 수정

CPU

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Discarding Fragments

- Alpha test를 이용한 잔디 렌더링 예 (cont.)
  - 최종 실행 결과



Discarding Fragments

- how grass is made in video games - YouTube

Blending

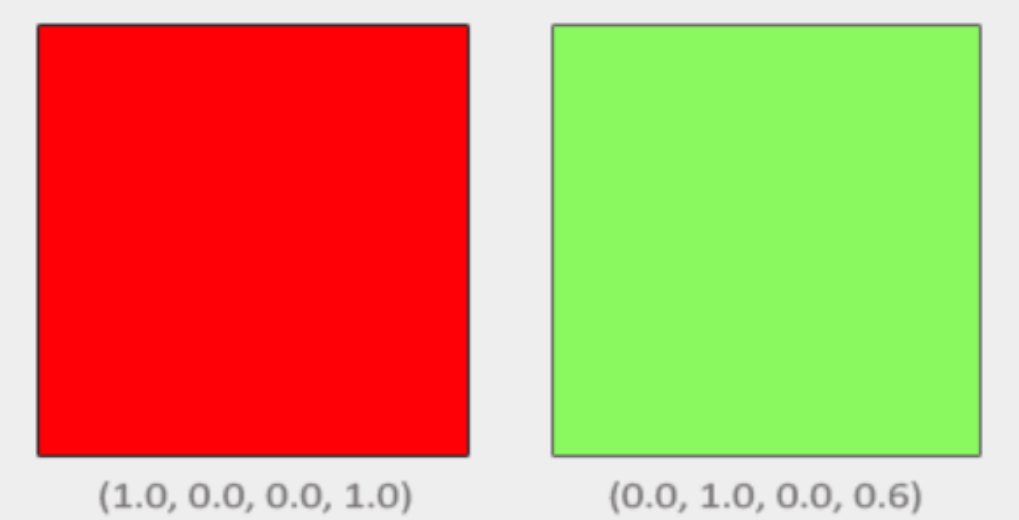
- 블렌딩은 반투명한 이미지를 표현할 때 사용하는 기법
- OpenGL에서 블렌딩을 사용하기 위해서는 해당 기능을 활성화해줘야 함 `glEnable(GL_BLEND);`
- 이후 아래 식에 따라 블렌딩 수행
  - 단, glBlendEquation() 설정에 따라 덧셈이 아니라 다른 연산을 수행할 수도 있음

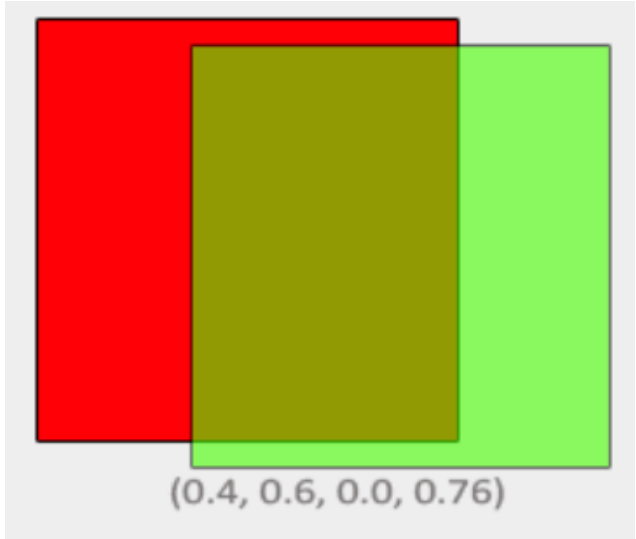
$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

- $\bar{C}_{source}$: the source color vector. This is the color output of the fragment shader.
- $\bar{C}_{destination}$: the destination color vector. This is the color vector that is currently stored in the color buffer.
- F_{source}: the source factor value. Sets the impact of the alpha value on the source color.
- $F_{destination}$: the destination factor value. Sets the impact of the alpha value on the destination color.

Blending

- Alpha값에 따른 블렌딩 예시





$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} * (1 - 0.6)$$

Blending Function

- glBlendFunc(GLenum sfactor, GLenum dfactor)
  - Source와 destination의 factor를 설정하는 함수
  - sfactor, dfactor에 설정 가능한 옵션 목록

Option	Value
GL_ZERO	Factor is equal to 0.
GL_ONE	Factor is equal to 1.
GL_SRC_COLOR	Factor is equal to the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_COLOR	Factor is equal to 1 minus the source color vector: $1 - \bar{C}_{source}$.
GL_DST_COLOR	Factor is equal to the destination color vector $\bar{C}_{destination}$
GL_ONE_MINUS_DST_COLOR	Factor is equal to 1 minus the destination color vector: $1 - \bar{C}_{destination}$.
GL_SRC_ALPHA	Factor is equal to the $alpha$ component of the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_ALPHA	Factor is equal to $1 - alpha$ of the source color vector $\bar{C}_{source}$.
GL_DST_ALPHA	Factor is equal to the $alpha$ component of the destination color vector $\bar{C}_{destination}$.
GL_ONE_MINUS_DST_ALPHA	Factor is equal to $1 - alpha$ of the destination color vector $\bar{C}_{destination}$.
GL_CONSTANT_COLOR	Factor is equal to the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_COLOR	Factor is equal to 1 - the constant color vector $\bar{C}_{constant}$.
GL_CONSTANT_ALPHA	Factor is equal to the $alpha$ component of the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_ALPHA	Factor is equal to $1 - alpha$ of the constant color vector $\bar{C}_{constant}$.

Constant color/alpha 값은
glBlendColor(red, green, blue, alpha)
함수로 설정 가능

Blending Function

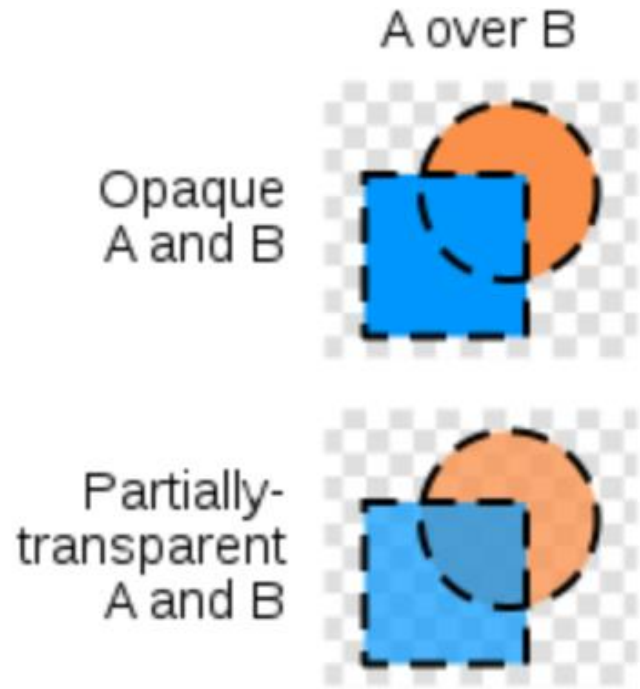
- glBlendFunc(GLenum sfactor, GLenum dfactor)
  - Alpha blending의 OVER 연산을 위한 가장 대표적인 사용 예시

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

$$\alpha_o = \alpha_a + \alpha_b(1 - \alpha_a)$$

$$C_o = \frac{C_a\alpha_a + C_b\alpha_b(1 - \alpha_a)}{\alpha_o}$$

Alpha compositing - Wikipedia



- glBlendFuncSeperate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha)
  - RGB 컬러와 Alpha별로 factor를 따로 설정하는 함수

```
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);
```

Blend Equation Function

- glBlendEquation `(GLenum mode)`
  - 블렌딩 시 사용되는 연산자(operator)를 설정하는 함수
  - 기본값은 GL_FUNC_ADD

- GL_FUNC_ADD: the default, adds both colors to each other: $\bar{C}_{result} = Src + Dst$.
- GL_FUNC_SUBTRACT: subtracts both colors from each other: $\bar{C}_{result} = Src - Dst$.
- GL_FUNC_REVERSE_SUBTRACT: subtracts both colors, but reverses order: $\bar{C}_{result} = Dst - Src$.
- GL_MIN: takes the component-wise minimum of both colors: $\bar{C}_{result} = min(Dst, Src)$.
- GL_MAX: takes the component-wise maximum of both colors: $\bar{C}_{result} = max(Dst, Src)$.

Rendering Semi-Transparent Textures

- 블렌딩 설정

CPU

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

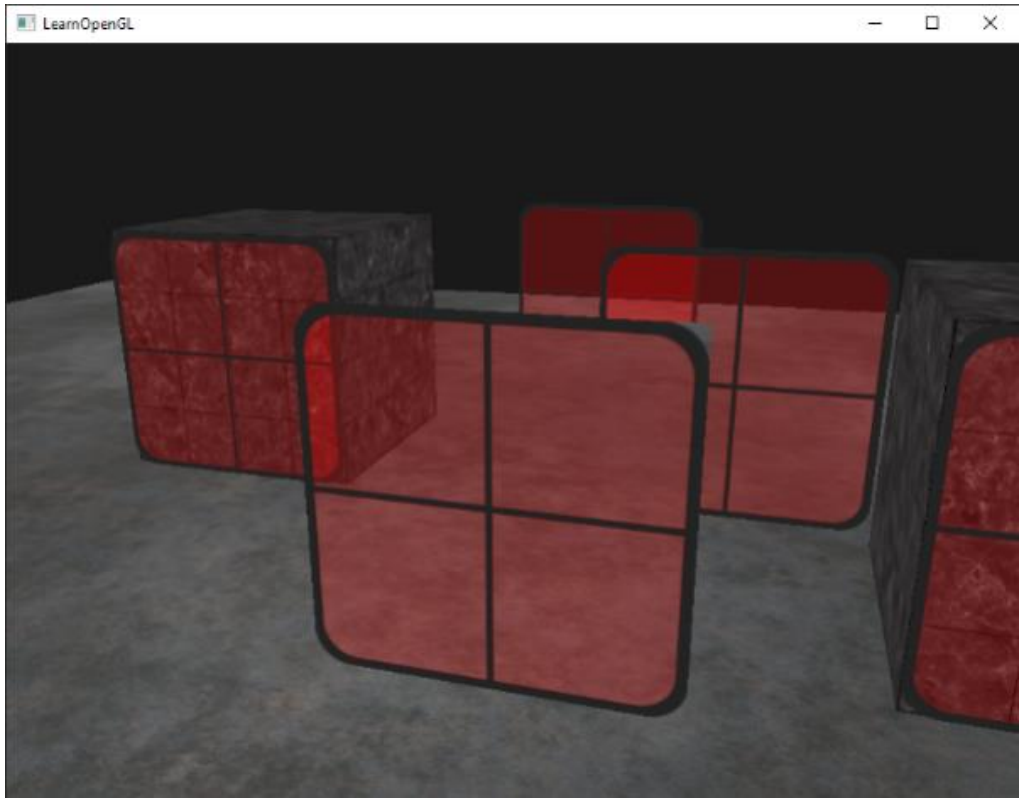
- Alpha test를 하지 않는 FS 버전으로 회귀

FS

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    FragColor = texture(texture1, TexCoords);
}
```

- 24p의 alpha값 0.75의 반투명 텍스처 사용

- 실행 결과
  - 일부 영역에서 앞과 뒤가 제대로 안 섞임
  - 투명도와 상관 없이 깊이검사를 수행했기 때문



Don’t Break the Order

- Back-to-front 순서로 렌더링하면 앞의 깊이 검사 문제를 해결 가능!
  - 불투명한 물체를 먼저 그림 (순서는 블렌딩과 관계 없으며, 일반적으로 front-to-back이 렌더링 효율을 높임)
  - 투명한 물체들을 back-to-front 순서로 정렬
  - 투명한 물체들을 위 순서에 따라 렌더링 (먼 곳의 물체를 먼저, 가까운 물체는 나중에 렌더링)

- 물체 정렬 방법
  - 시점으로부터 물체까지의 거리를 얻은 후, 이를 STL 라이브러리의 map 자료 구조에 저장
    (map 자료구조는 key값을 기반으로 오름차순 정렬을 자동으로 수행하며, 자바의 TreeMap과 동일)
  - 거리값을 key로 설정, 윈도우의 위치를 value로 설정하면 거리가 짧은 물체에서 긴 물체 순으로 저장

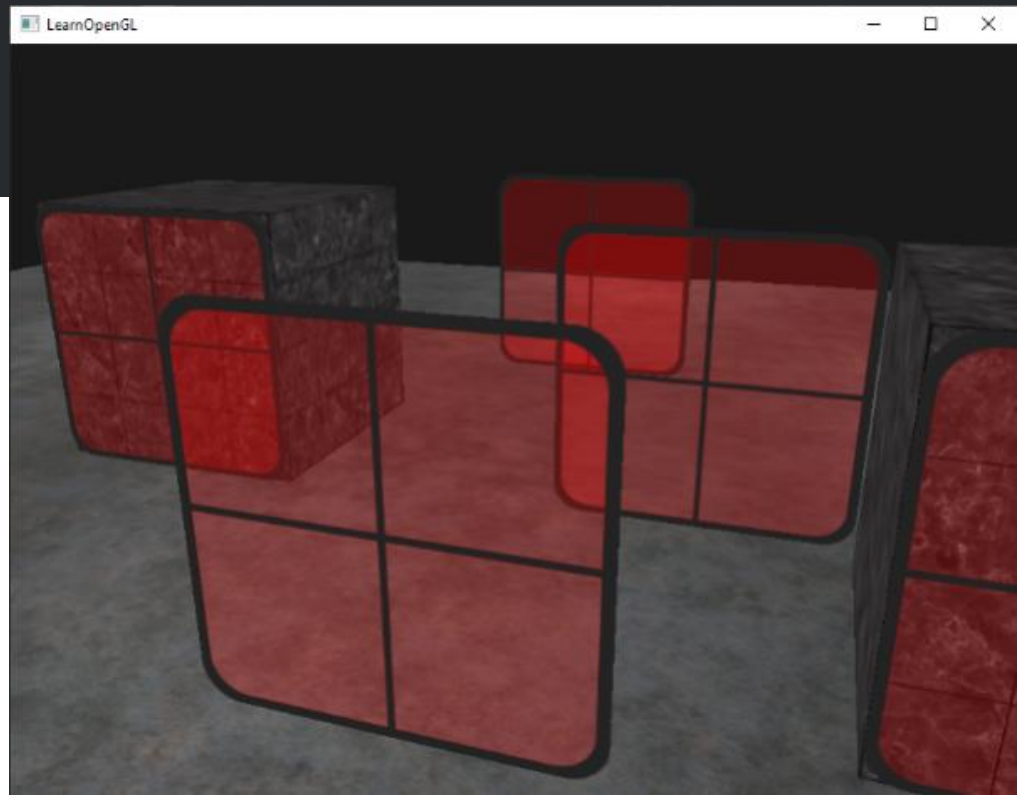
```
std::map<float, glm::vec3> sorted;
for (unsigned int i = 0; i < windows.size(); i++)
{
    float distance = glm::length(camera.Position - windows[i]);
    sorted[distance] = windows[i];
}
```

Don’t Break the Order

- 물체 정렬 방법 (cont.)
  - Back-to-front order로 물체를 렌더링해야 하므로, 정렬된 역순으로 iterator를 사용하여 렌더링

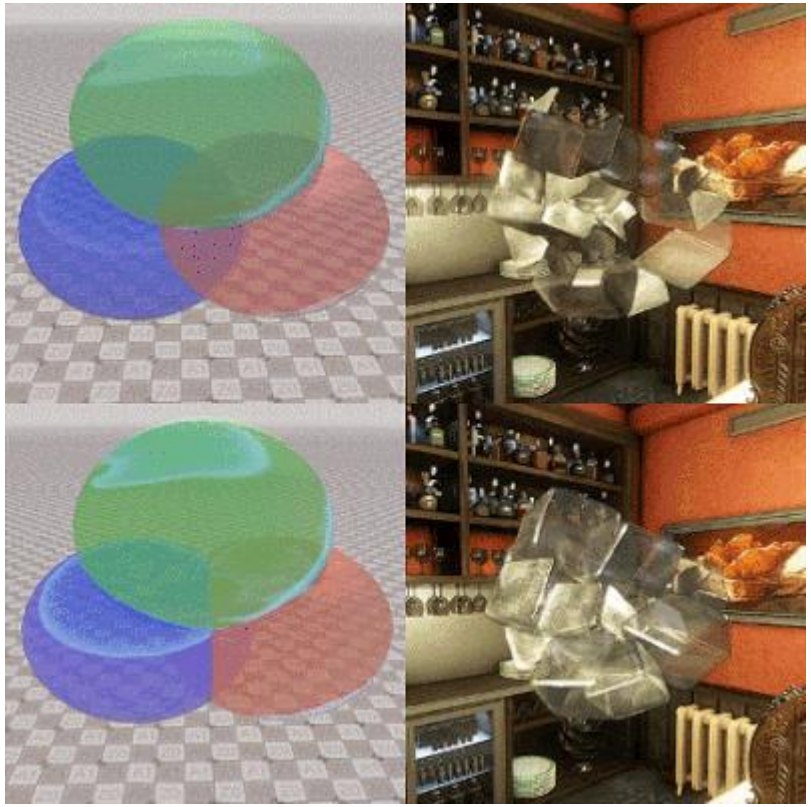
```
for(std::map<float,glm::vec3>::reverse_iterator it = sorted.rbegin(); it != sorted.rend(); ++it)
{
    model = glm::mat4(1.0f);
    model = glm::translate(model, it->second);
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

- 제대로 렌더링 된 결과 화면



OIT (Order-Independent Transparency)

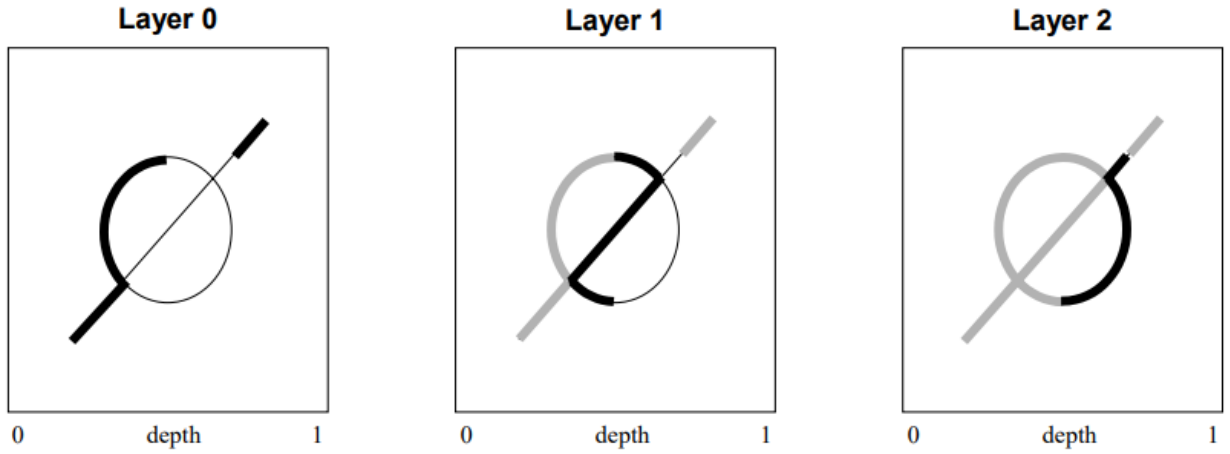
- 물체들을 렌더링하는 순서와 관계 없이 반투명 효과를 낼 수 있는 기술
- OIT가 필요한 예
  - 삼각형들이 겹쳐 그려지는 오목한 지오메트리 (유리, 와인잔, 유리 조각품 등)
  - 서로 교차하는 지오메트리 (수많은 머리카락)
  - 투명한 물체 안의 투명한 물체 (유리잔 안의 액체)
  - 연기 및 구름

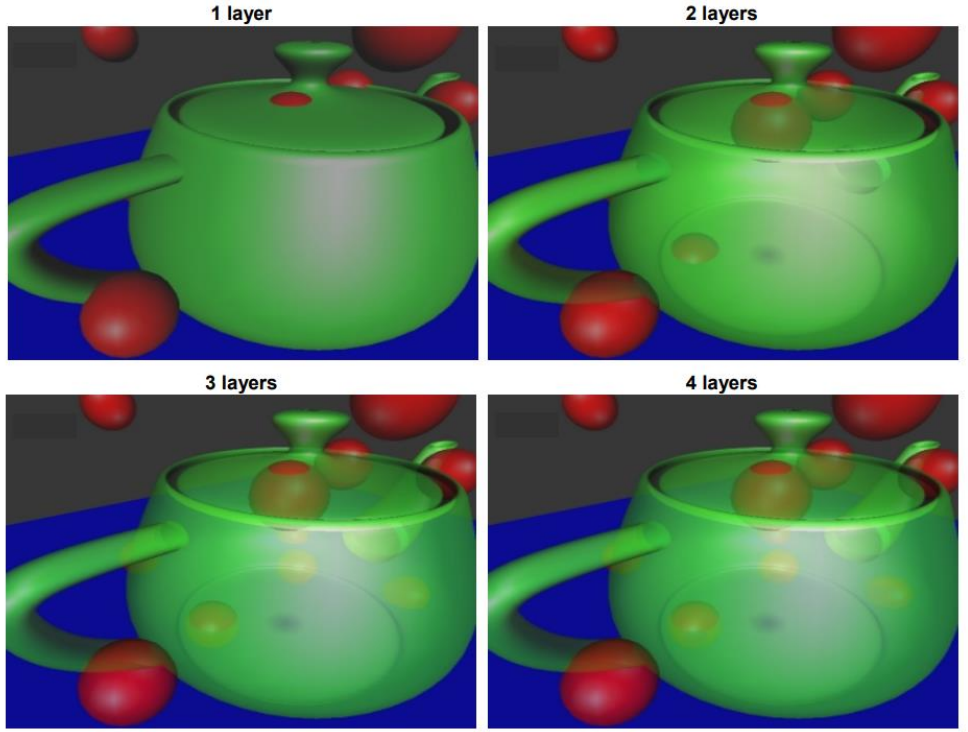


OIT(Order Independent Transparency) - Lumberyard 사용 설명서 (amazon.com)

OIT (Order-Independent Transparency)

- Depth peeling
  - 마치 양파 껍질을 까듯이 depth layer별로 여러 번 렌더링을 수행하여, 각 layer별 결과를 메모리에 저장
  - Fragment 단위로 back-to-front 순서를 알게 되었으므로, 이에 따라 OVER연산으로 blending 수행
  - 지정한 layer 개수 안에서는 정확한 결과 도출
  - Layer 개수만큼 렌더 패스 필요 → 성능 대폭 저하 가능성





OrderIndependentTransparency.pdf (nvidia.com)

OIT (Order-Independent Transparency)

- Weighted, blended order-independent transparency
  - 아주 정확하지는 않지만, 속도가 빠르고 결과가 그럴듯한 근사(approximation) 방법
  - 첫번째 패스에서, 각 객체를 렌더링하면서 계산되는 아래 식의 누적합과 누적곱을 두 텍스처에 저장 (glBlendFunci() 함수를 이용)
  - 각 fragment의 depth값에 따라 weight를 다르게 계산하여 이를 blending factor 중 하나로 사용하는 것이 핵심
  - 이 두 텍스처를 이용하여 다음 렌더 패스에서 compositing 수행 → 단 2번의 렌더 패스만 필요
  - 필요시 guest article 내 설명 및 구현 소스 코드 참조 LearnOpenGL - Weighted Blended

Weight

accumTexture.rgb

"Coverage"

Weighted Blended Order-Independent Transparency (JCGT)

$$C_f = \frac{\sum_{i=1}^{n} C_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} \alpha_i \cdot w(z_i, \alpha_i)} \left(1 - \prod_{i=1}^{n}(1-\alpha_i)\right) + C_0 \prod_{i=1}^{n}(1-\alpha_i)$$

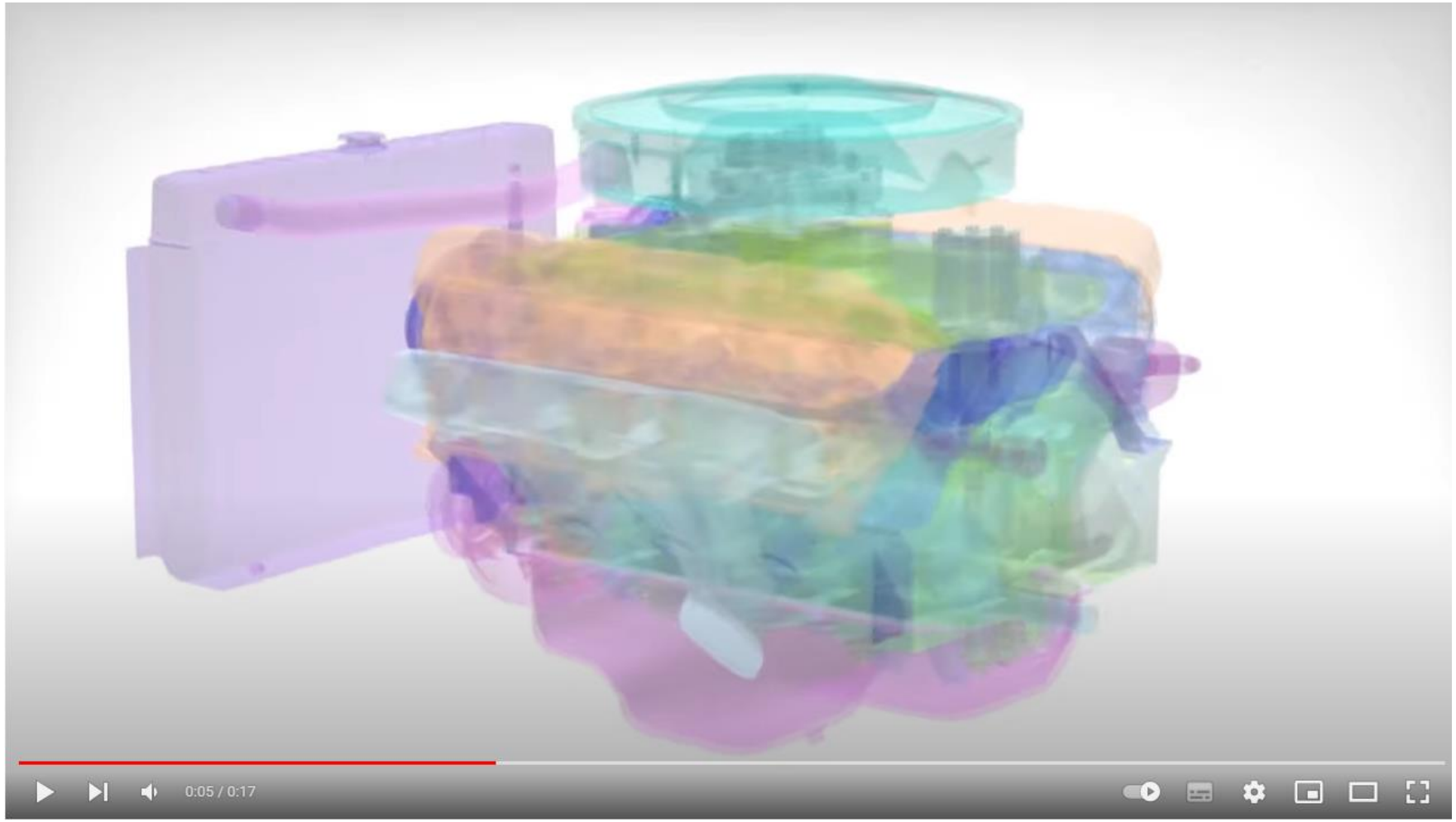
accumTexture.a

"Revealage"

revealageTexture.r

OIT (Order-Independent Transparency)

- Weighted, Blended Order-Independent Transparency: Engine Scene - YouTube



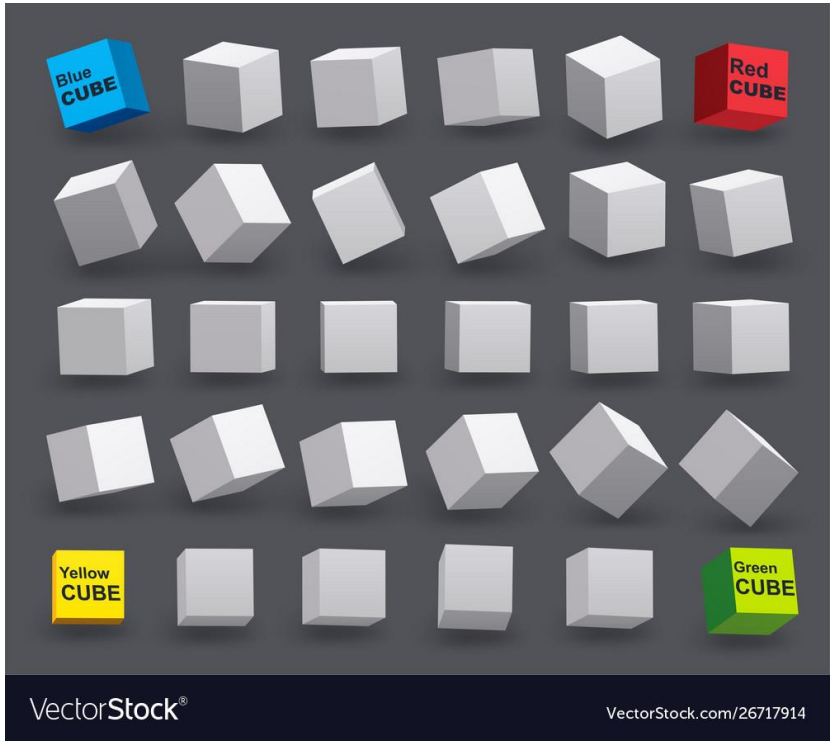
ⓒ nvidia

Face Culling

Face Culling

- 3D 큐브에서 보이는 면을 상상해 보면..
  - 현재 시점에서 보이는 면은 1~3개임
  - 따라서 6면체의 50% 이상은 렌더링을 하지 않고 계산 절약 가능!

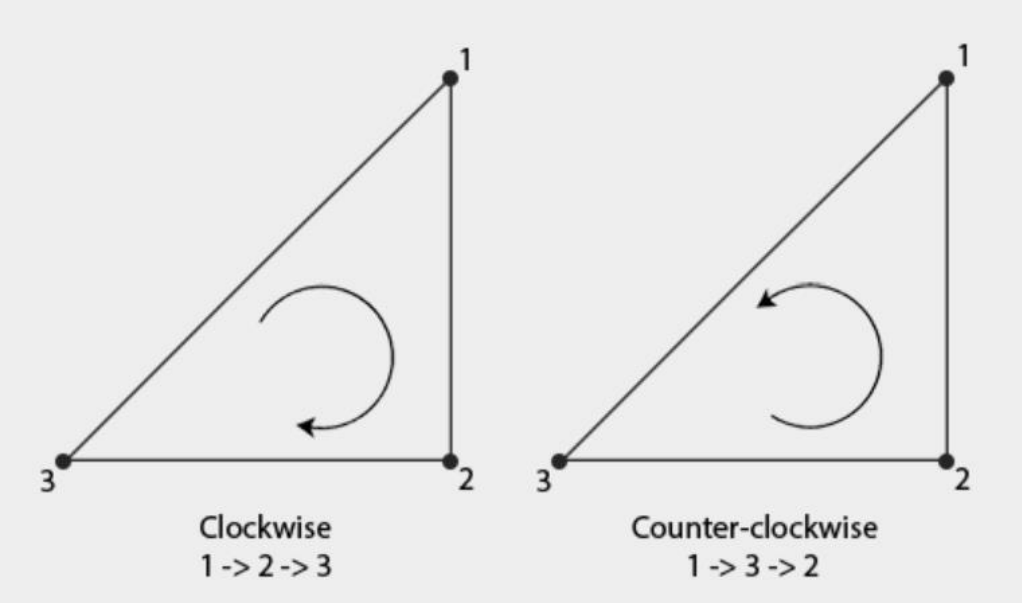
- Face culling
  - Viewer의 관점에서 앞을 향하는(front facing) 면만 렌더링하고, 뒤를 향하는 (back facing) 면은 폐기 가능 (back-face culling)
  - 렌더링 성능을 향상시키는 좋은 방법
  - 물론, 반투명한 물체에서는 이 culling을 비활성화해야 함

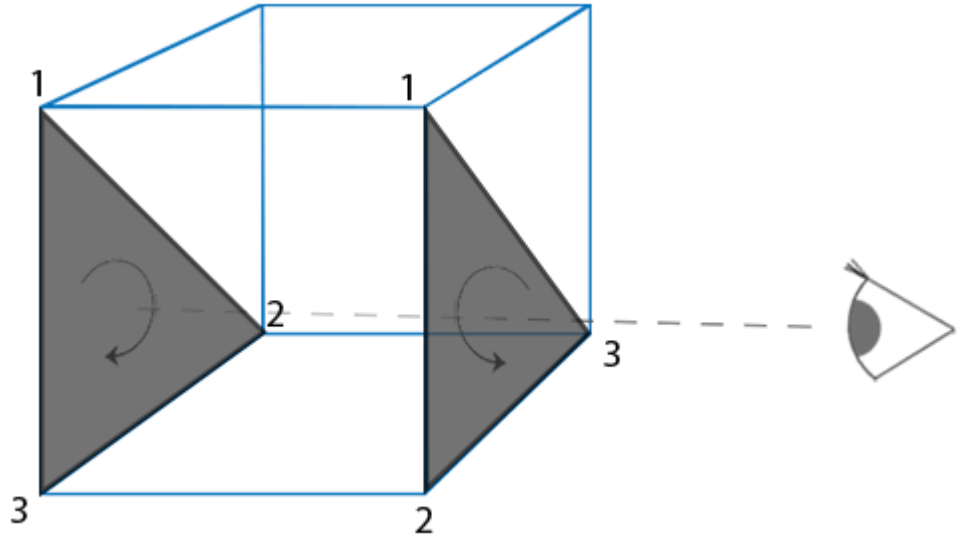


Set white cubes in various tilt angles Royalty Free Vector (vectorstock.com)

Winding Order

- 삼각형 내 정점을 감는 순서
  - 시계 방향 (clockwise) 또는 반시계 방향 (counter-clockwise) 중 어떤 것이 앞면을 향하는지 선택
  - OpenGL에서는 일반적으로 반시계 방향 사용
  - 삼각형의 정점 데이터 입력시, 이 삼각형은 앞면을 향하고 있다고 가정하고 정점을 입력해야 함
  - 삼각형의 실제 winding order는 VS가 실행된 후 rasterization 단계에서 계산
    (Viewer의 입장에서 이 순서를 가지고 앞면 또는 뒷면을 판단 가능하게 됨)



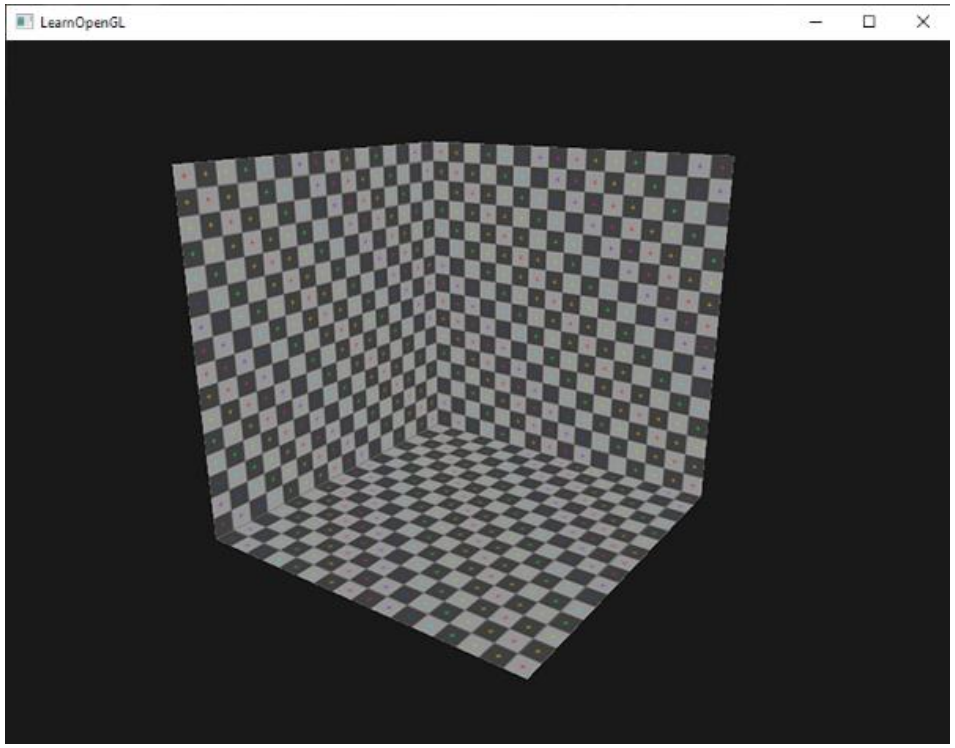


Face Culling in OpenGL

- Face culling 활성화 `glEnable(GL_CULL_FACE);`
- Cull할 face 설정 `glCullFace(GL_FRONT);`
  - GL_BACK, GL_FRONT, 또는 GL_FRONT_AND_BACK
- 앞을 향하는 면의 정점이 감기는 방향 설정
  `glFrontFace(GL_CCW);`
  - GL_CCW 또는 GL_CW
- 위와 똑같은 기능을 하는 명령

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glFrontFace(GL_CW);
```

- 큐브의 back face만 그리도록 한 예



© nvidia

마무리

마무리

- Advanced OpenGL의 첫 번째 시간으로, 아래와 같은 내용을 살펴보았습니다.
  - Depth testing
  - Stencil testing
  - Alpha testing
  - Alpha blending
  - (Back)-face culling

- 다음 시간은 중간 고사 전 마지막 수업으로, 아래 실습을 수행할 예정입니다.
  - 기반 코드에서, 아래 코드를 참조하여 alpha testing 및 alpha blending 효과 적용
    Code Viewer. Source code: src/4.advanced_opengl/3.1.blending_discard/blending_discard.cpp (learnopengl.com)
    Code Viewer. Source code: src/4.advanced_opengl/3.2.blending_sort/blending_sorted.cpp (learnopengl.com)
  - 여기에 아래 코드를 참조하여 object outlining 효과 추가
    Code Viewer. Source code: src/4.advanced_opengl/2.stencil_testing/stencil_testing.cpp (learnopengl.com)