Advanced OpenGL (4)

GPU Programming

2022학년도
2학기

ⓒ nvidia

Instancing

Instancing

- Instancing의 필요성
  - 동일한 vertex data set을 가지지만, 서로 다른 world transformation을 가지는 수많은 model이 있다면?
    예) 잔디로 구성된 scene
  - Model의 수만큼 draw call(glDrawArrays, glDrawElements 등)을 호출해야 함
    (수천개의 잔디 → 수천번의 draw call)

```
for(unsigned int i = 0; i < amount_of_models_to_draw; i++)
{
    DoSomePreparations(); // bind VAO, bind textures, set uniforms etc.
    glDrawArrays(GL_TRIANGLES, 0, amount_of_vertices);
}
```

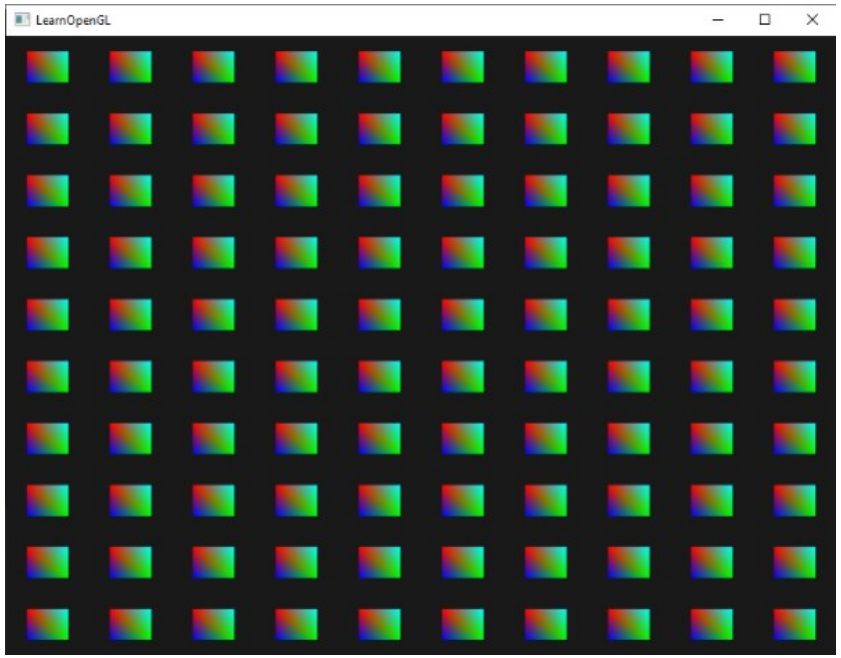
  - Draw call의 증가는 성능 병목으로 이어질 수 있음
    (OpenGL은 vertex data를 그리기 전에 여러가지 준비 작업을 해야 하고,
    이 결과는 상대적으로 느린 CPU→GPU 버스를 통해 전달되기 때문)

- Instancing의 정의 및 이점
  - 한 번의 draw call로 여러 개의 object를 그리는데 필요한 data를 GPU로 보내는 방법
  - CPU overhead 뿐 아니라 CPU에서 GPU로의 통신을 크게 줄일 수 있음

Instancing

- Instancing의 사용 방법
  - glDrawArrays() → glDrawArraysInstanced()
  - glDrawElements() → glDrawElementsInstanced()
  - 추가적으로 instance의 개수가 파라미터에 포함
  - 각 instance는 vertex shader에서 gl_InstanceID라는 built-in 변수로 구분 (ID는 0부터 시작)

- 100개의 사각형 instance을 그리는 예제
  - 100개의 offset vector를 저장하는 uniform array를 indexing하여 각 사각형의 위치를 계산



Instancing

- 100개의 사각형 instance을 그리는 예제 (cont.)

VS

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out vec3 fColor;

uniform vec2 offsets[100];

void main()
{
    vec2 offset = offsets[gl_InstanceID];
    gl_Position = vec4(aPos + offset, 0.0, 1.0);
    fColor = aColor;
}
```

FS

```
#version 330 core
out vec4 FragColor;

in vec3 fColor;

void main()
{
    FragColor = vec4(fColor, 1.0);
}
```

Instancing

- 100개의 사각형 instance을 그리는 예제 (cont.)

```
float quadVertices[] = {
    // positions     // colors
    -0.05f,  0.05f,  1.0f, 0.0f, 0.0f,
     0.05f, -0.05f,  0.0f, 1.0f, 0.0f,
    -0.05f, -0.05f,  0.0f, 0.0f, 1.0f,

    -0.05f,  0.05f,  1.0f, 0.0f, 0.0f,
     0.05f, -0.05f,  0.0f, 1.0f, 0.0f,
     0.05f,  0.05f,  0.0f, 1.0f, 1.0f
};
```

CPU

```
glm::vec2 translations[100];
int index = 0;
float offset = 0.1f;
for(int y = -10; y < 10; y += 2)
{
    for(int x = -10; x < 10; x += 2)
    {
        glm::vec2 translation;
        translation.x = (float)x / 10.0f + offset;
        translation.y = (float)y / 10.0f + offset;
        translations[index++] = translation;
    }
}
```

```
shader.use();
for(unsigned int i = 0; i < 100; i++)
{
    shader.setVec2(("offsets[" + std::to_string(i) + "]"), translations[i]);
}
```

```
glBindVertexArray(quadVAO);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);
```

Instanced Arrays

- Offset을 uniform 형태로 전달하는 앞 예제에서 만약 엄청나게 많은 instance를 그린다면?
  - GPU에서 사용 가능한 uniform data의 제약에 걸릴 수 있음 (GL_MAX_UNIFORM_LOCATIONS)

- Instanced arrays를 사용하면 이러한 제약을 극복 가능
  - Uniform 변수 대신, vertex shader의 입력 형태로 array를 전달
  - GPU는 각 vertex shader가 시작될 때마다 현재 vertex에 속해 있는 vertex attribute를 검색하는데, instanced arrays의 경우 이와 달리 새로운 instance가 그려질 때마다 update되는 vertex attribute로 정의

- Instanced array의 사용 방법
  - gl_InstanceID 대신에 aOffset을 직접 사용

VS

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aOffset;

out vec3 fColor;

void main()
{
    gl_Position = vec4(aPos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

Instanced Arrays

- Instanced array의 사용 방법 (cont.)
  - VBO에 offset 배열(앞 예제의 translations)을 저장하고 attribute pointer를 이에 맞게 설정

```
unsigned int instanceVBO;
glGenBuffers(1, &instanceVBO);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec2) * 100, &translations[0], GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

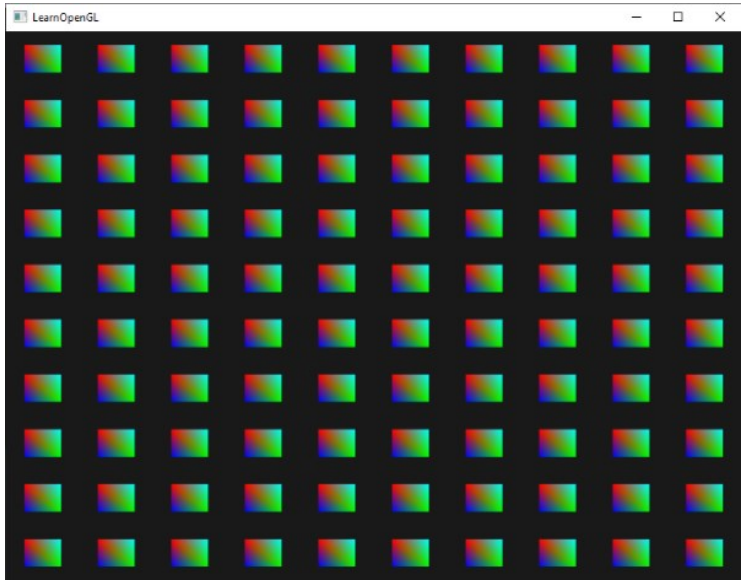
```
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glVertexAttribDivisor(2, 1);
```

CPU

- glVertexAttribDivisor(index, divisor)
  - index: vertex attribute의 index (VS의 location)
  - divisor 0 (기본값): VS의 각 iteration마다 vertex attribute의 내용을 갱신
  - divisor 1: 새로운 instance를 시작할 때 vertex attribute의 내용을 갱신
  - divisor n: n개 instance마다 vertex attribute의 내용을 갱신 (=n개 instance가 같은 값을 공유)

Instanced Arrays

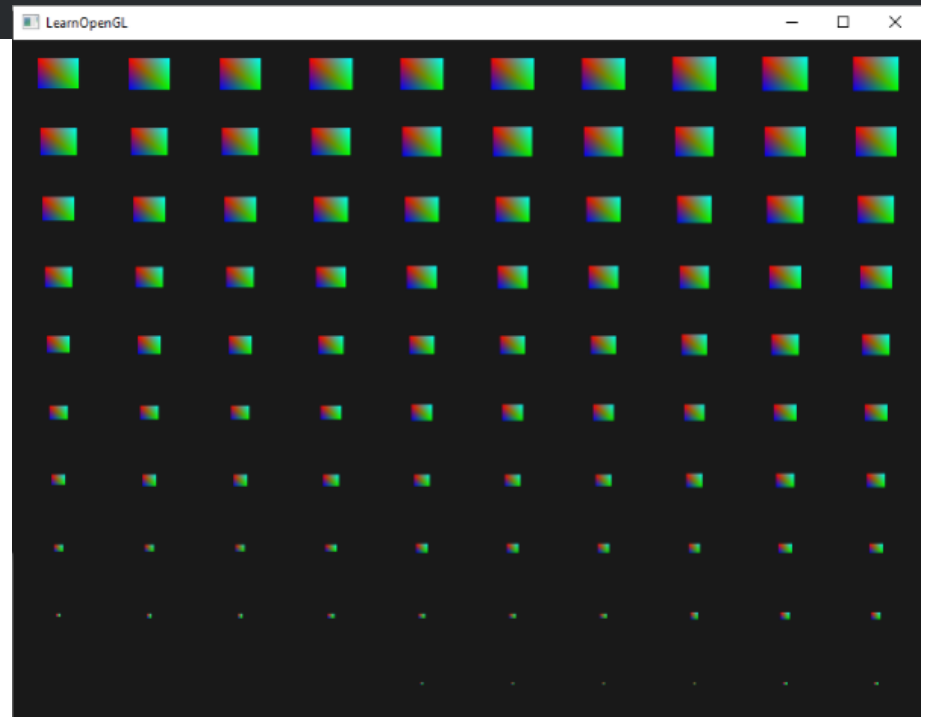
- Instanced array 실행 결과
  - Instance의 구성이 바뀌지 않았으므로, Uniform array 사용시와 결과는 같음



- 우상단에서 좌하단으로 사각형의 크기가 downscale되도록 변경

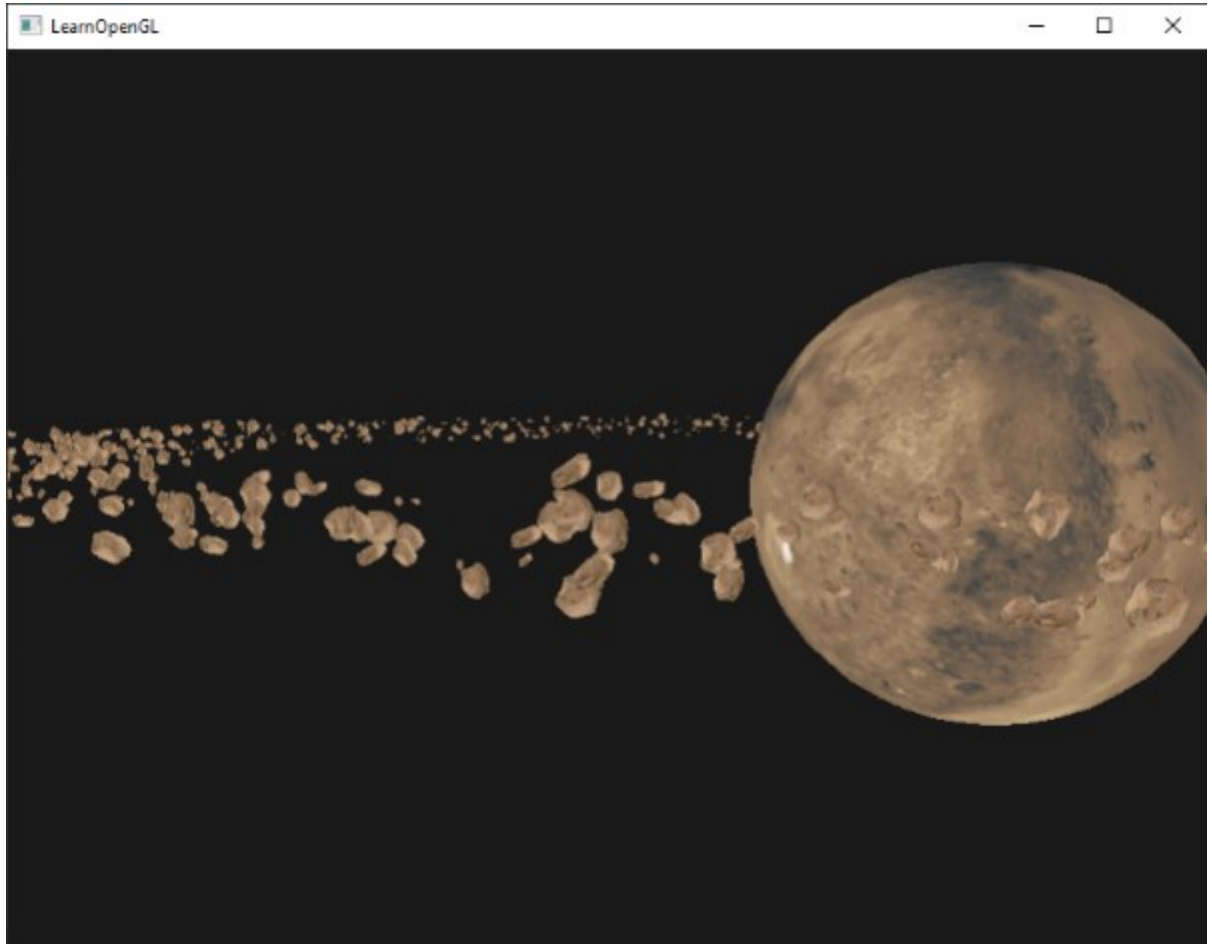
```
void main()
{
    vec2 pos = aPos * (gl_InstanceID / 100.0);
    gl_Position = vec4(pos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

VS



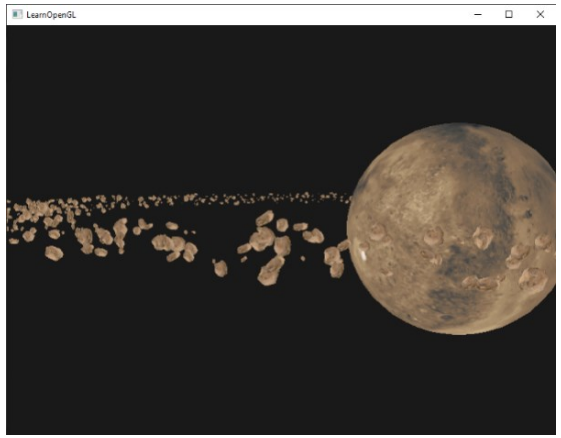
An Asteroid Field

- 하나의 큰 행성 주위에 소행성 고리가 존재하는 scene
  - 소행성 고리는 수천 또는 수만개의 암석을 포함
  - 각 소행성은 단일 model로 표현 가능
  - 각 소행성은 고유의 transformation matrix를 가짐
  - Instanced rendering이 특히 유용한 예



An Asteroid Field

- 각 소행성별 model transformation matrix 초기화
  - 소행성은 xz 평면 상에 원 형태로 배치
  - Translation시 고리가 자연스럽게 보이도록, 랜덤한 offset별 변위(displacement) 추가
  - 랜덤 scale과 rotation 추가
  - 각 소행성의 위치, 크기, 회전 각도 모두 불규칙적으로 정해짐



CPU

```
unsigned int amount = 1000;
glm::mat4 *modelMatrices;
modelMatrices = new glm::mat4[amount];
srand(glfwGetTime()); // initialize random seed
float radius = 50.0;
float offset = 2.5f;
for(unsigned int i = 0; i < amount; i++)
{
    glm::mat4 model = glm::mat4(1.0f);
    // 1. translation: displace along circle with 'radius' in range [-offset, offset]
    float angle = (float)i / (float)amount * 360.0f;
    float displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float x = sin(angle) * radius + displacement;
    displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float y = displacement * 0.4f; // keep height of field smaller compared to width of x and z
    displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float z = cos(angle) * radius + displacement;
    model = glm::translate(model, glm::vec3(x, y, z));

    // 2. scale: scale between 0.05 and 0.25f
    float scale = (rand() % 20) / 100.0f + 0.05;
    model = glm::scale(model, glm::vec3(scale));

    // 3. rotation: add random rotation around a (semi)randomly picked rotation axis vector
    float rotAngle = (rand() % 360);
    model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

    // 4. now add to list of matrices
    modelMatrices[i] = model;
}
```

An Asteroid Field

- Instance 없이 각각의 소행성을 렌더링
  - amount값을 늘리면 늘릴수록 그에 비례하여 FPS가 떨어짐 (Fraps 프로그램을 설치하여 확인 가능)

CPU

```
// draw planet
shader.use();
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, -3.0f, 0.0f));
model = glm::scale(model, glm::vec3(4.0f, 4.0f, 4.0f));
shader.setMat4("model", model);
planet.Draw(shader);

// draw meteorites
for(unsigned int i = 0; i < amount; i++)
{
    shader.setMat4("model", modelMatrices[i]);
    rock.Draw(shader);
}
```

An Asteroid Field

- Instanced rendering 방식 사용
  - 4x4 instancedMatrix를 instanced array 형태로 입력

VS

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in mat4 instanceMatrix;

out vec2 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    gl_Position = projection * view * instanceMatrix * vec4(aPos, 1.0);
    TexCoords = aTexCoords;
}
```

An Asteroid Field

- Instanced rendering 방식 사용 (cont.)
  - modelMatrices를 VBO를 통해 GPU로 넘겨주고, VAO에서 matrix data에 대한 vertex attribute pointer 설

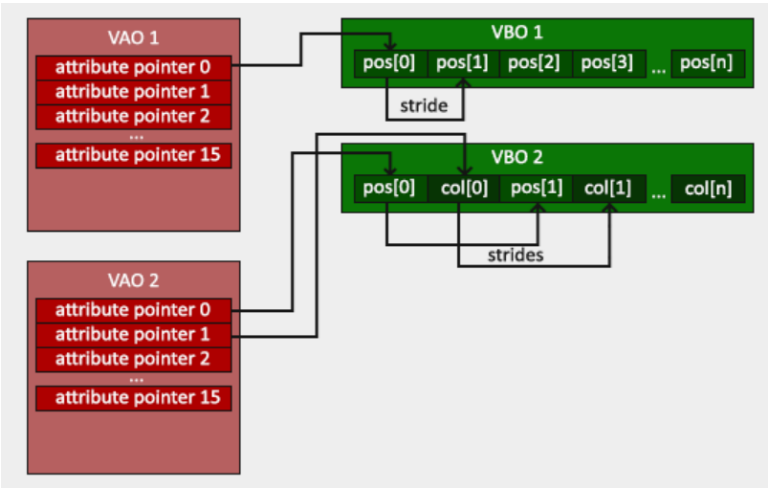
CPU

```
// vertex buffer object
unsigned int buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, amount * sizeof(glm::mat4), &modelMatrices[0], GL_STATIC_DRAW);

for(unsigned int i = 0; i < rock.meshes.size(); i++)
{
    unsigned int VAO = rock.meshes[i].VAO;
    glBindVertexArray(VAO);
    // vertex attributes
    std::size_t vec4Size = sizeof(glm::vec4);
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)0);
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(1 * vec4Size));
    glEnableVertexAttribArray(5);
    glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(2 * vec4Size));
    glEnableVertexAttribArray(6);
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(3 * vec4Size));

    glVertexAttribDivisor(3, 1);
    glVertexAttribDivisor(4, 1);
    glVertexAttribDivisor(5, 1);
    glVertexAttribDivisor(6, 1);

    glBindVertexArray(0);
}
```



An Asteroid Field

- Instanced rendering 방식 사용 (cont.)
  - glDrawElementsInstanced() 함수를 이용하여 렌더링
  - 아래 rock 내부의 meshes의 크기는 1이므로, loop는 한 번만 돎
  - Model class 내의 draw() 함수를 호출하지 않으므로, 수동으로 texture binding을 해 줘야 함

CPU

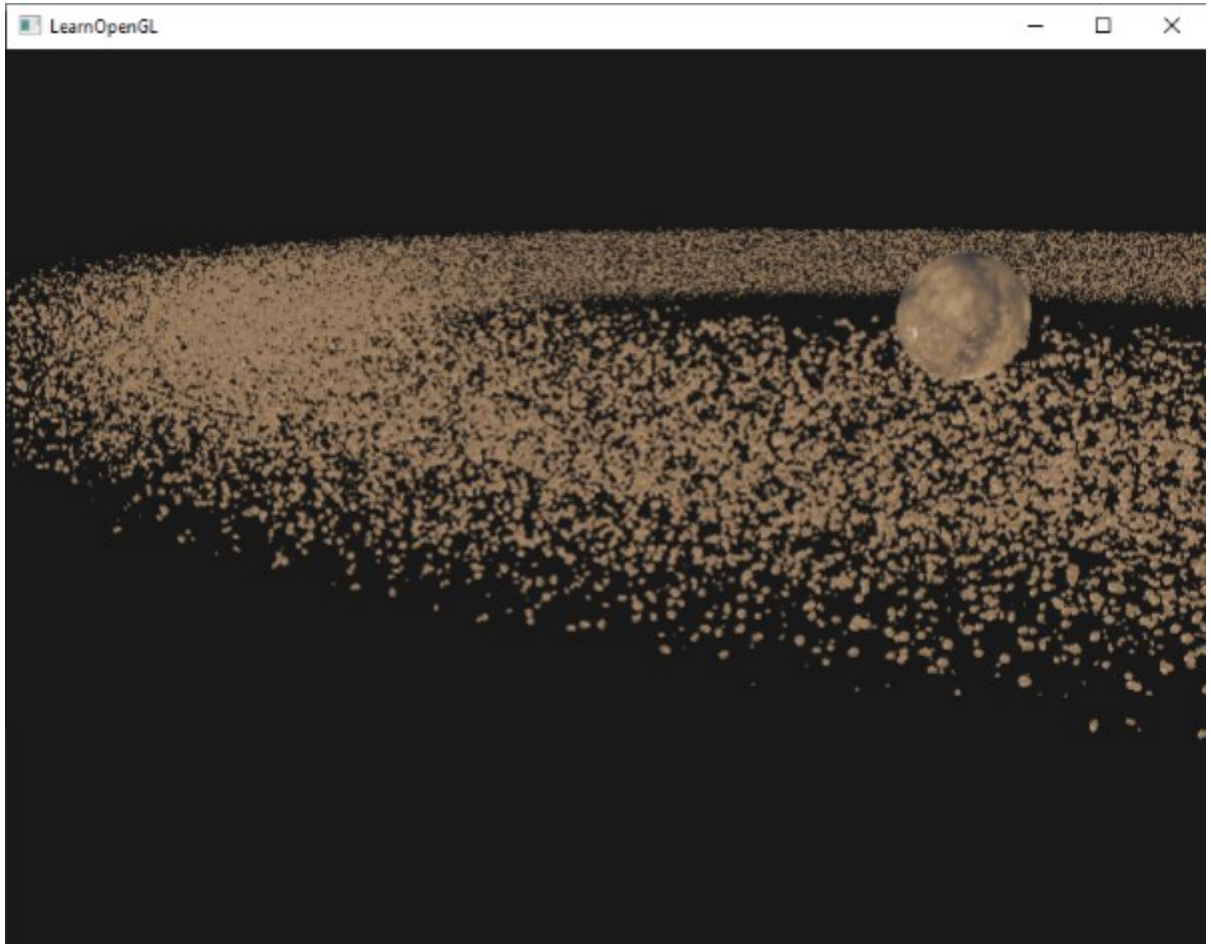
```
// draw meteorites
asteroidShader.use();
asteroidShader.setInt("texture_diffuse1", 0);

glActiveTexture(GL_TEXTURE0);
// note: we also made the textures_loaded vector public (instead of private) from the model class.
glBindTexture(GL_TEXTURE_2D, rock.textures_loaded[0].id);

for (unsigned int i = 0; i < rock.meshes.size(); i++)
{
    glBindVertexArray(rock.meshes[i].VAO);
    glDrawElementsInstanced(GL_TRIANGLES, static_cast<unsigned int>(rock.meshes[i].indices.size()), GL_UNSIGNED_INT, 0, amount);
    glBindVertexArray(0);
}
```

An Asteroid Field

- Instanced rendering 방식 사용 (cont.)
  - 100,000만개 소행성도 거뜸 없이 렌더링
  - 이 방식은 잔디, 꽃, particle, 소행성 등 반복되는 shape들을 렌더링하는 데 널리 쓰임

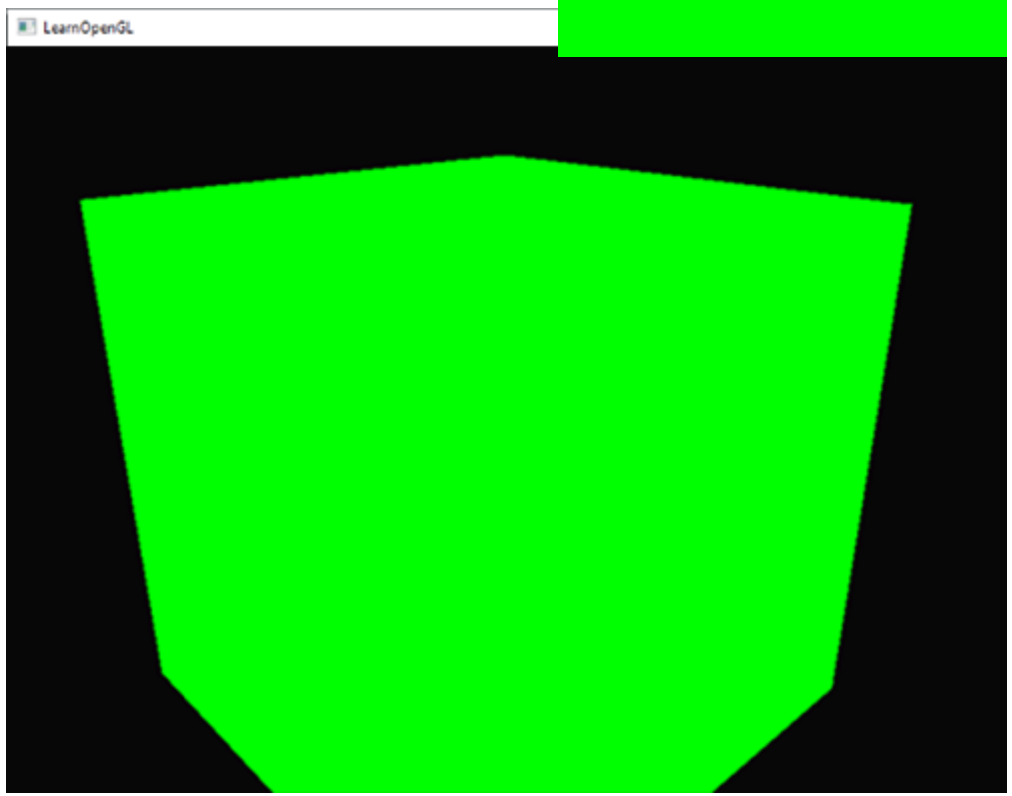


ⓒ nvidia

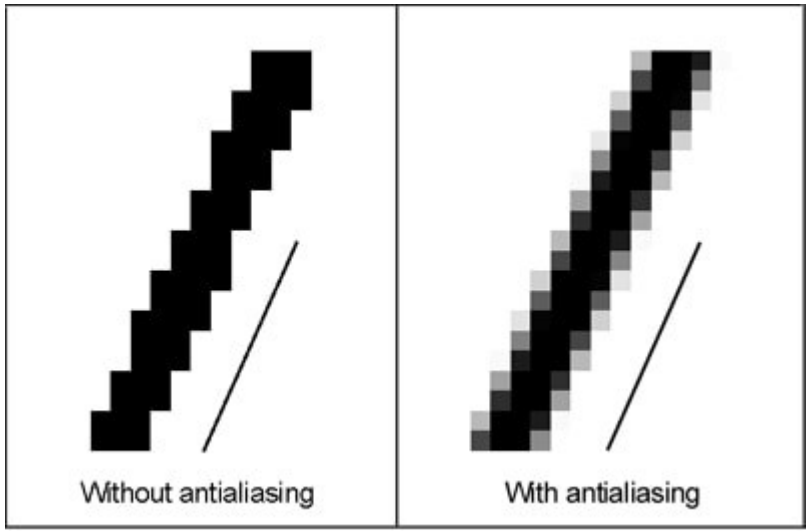
Anti Aliasing

Anti Aliasing

- Aliasing
  - 오른쪽과 같은 edge 상의 jagged pattern
  - Rasterizer가 vertex data를 해상도에 맞게 fragment로 바꾸는 과정에서 발생



- Anti-aliasing
  - 위와 같은 픽셀 단위의 계단 현상을 없애 주는 기법들

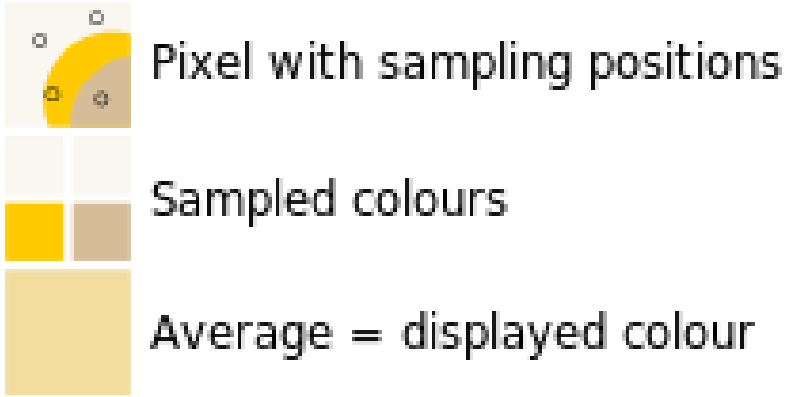


What Is The Best Anti-Aliasing Mode? [Simple] - DisplayNinja

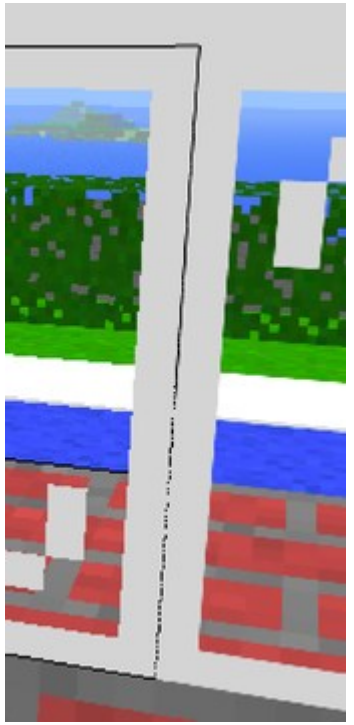
Super Sample Anti-Aliasing (SSAA)

- 가장 품질이 좋지만, 가장 연산량이 많은 AA 기법

- 픽셀당 많은 fragment 샘플을 취한 후, 샘플별로 컬러를 계산한 후 이를 평균 냄
  - 계산해야 하는 fragment shading 연산은 픽셀당 샘플 수에 비례하여 증가 (4x SSAA = 4x fragments)



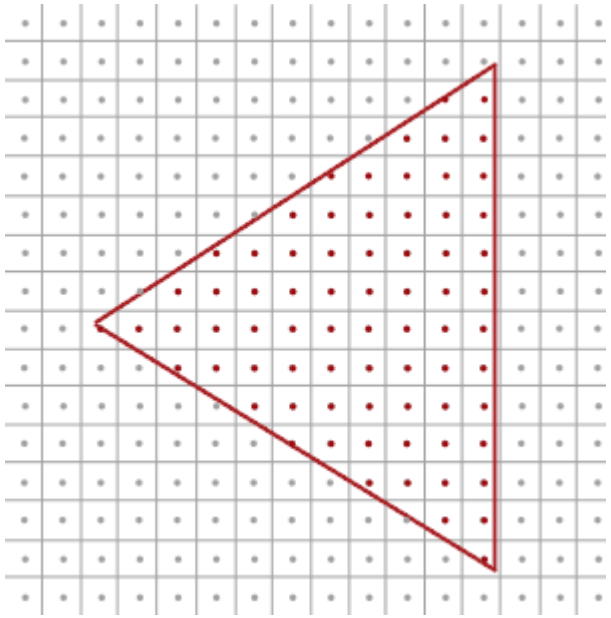
Supersampling - Wikipedia

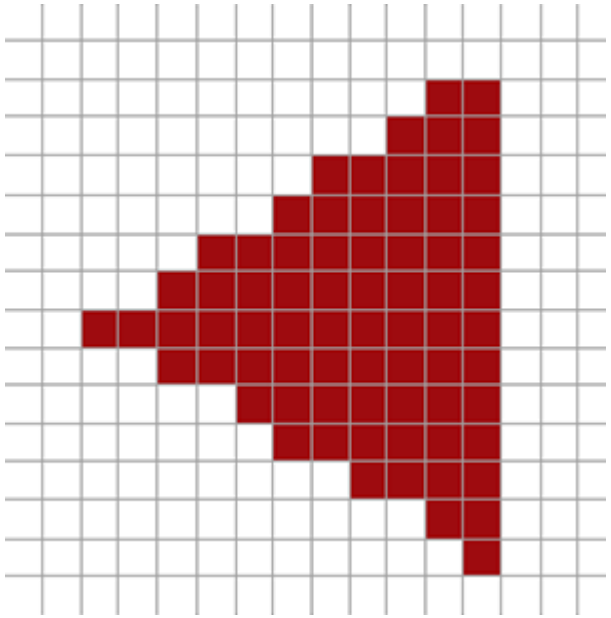


Multisampling

- Rasterizer
  - 최종적으로 처리된 vertex들과 fragment shader 사이에 있는 모든 알고리즘과 프로세스의 결합
  - 각 vertex는 어떠한 좌표라도 가질 수 있지만, fragment는 해상도에 제한된 좌표만 가질 수 있음
  - Vertex와 fragment의 좌표 간 1:1 매핑은 불가능하므로, rasterizer는 이 매핑 방법을 결정해야 함

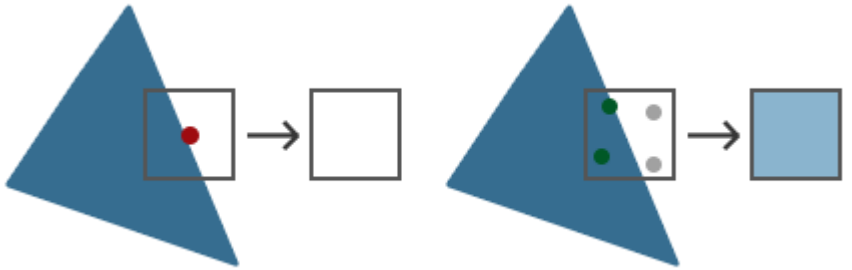
- Rasterizer의 fragment 처리 과정
  - 픽셀의 정 중앙을 샘플 point로 취해,
    해당 fragment가
    삼각형 내부에 존재하는지 아닌지 결정
  - 위 결과에 따라 fragment shading 수행
  - 이 과정에서 edge 부분에
    aliasing이 필연적으로 발생
    (화면 내 픽셀의 개수가 한정되었기 때문)



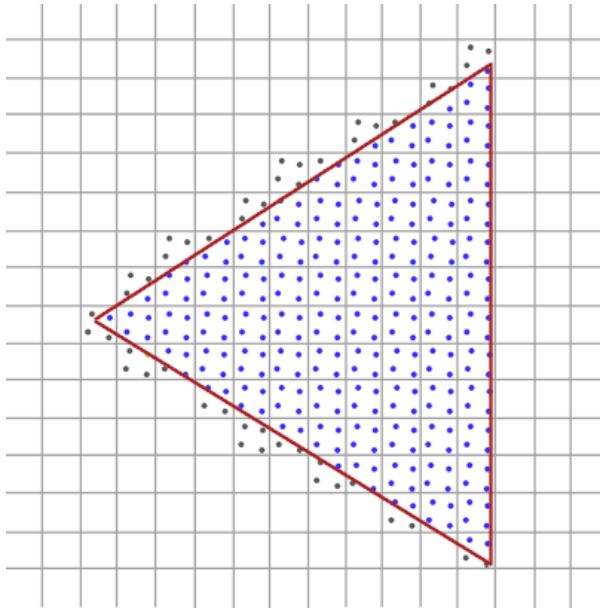


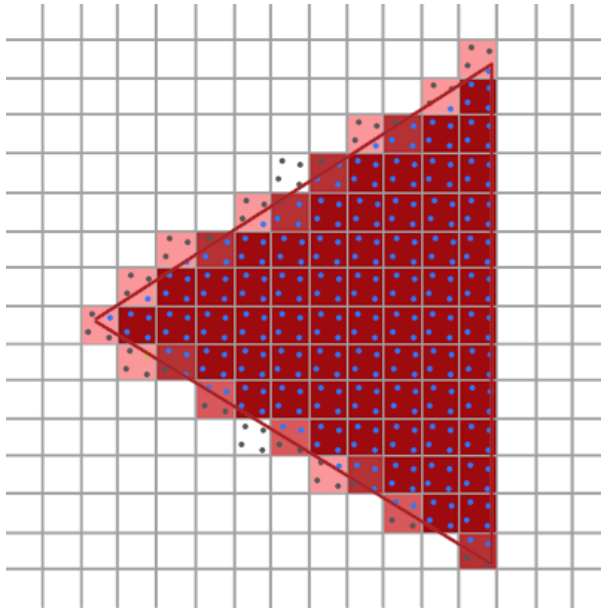
Multisampling

- Multisample Anti-Aliasing(MSAA)
  - 픽셀당 여러 개의 샘플 point들, 즉 subsample을 사용
  - Fragment의 색상을 결정할 때, 이 fragment의 subsample들이 현재 그리고 있는 primitive를 얼마만큼 cover하는지 계산
  - 픽셀 중앙을 기준으로 수행된 fragment shader의 계산 결과값을 이 coverage만큼만 fragment에 반영
  - SSAA와 달리 subsample수와 관계 없이 shading 연산이 fragment당 1번만 계산됨
  - Color/depth/stencil buffer의 크기가 subsample의 수에 비례하여 증가



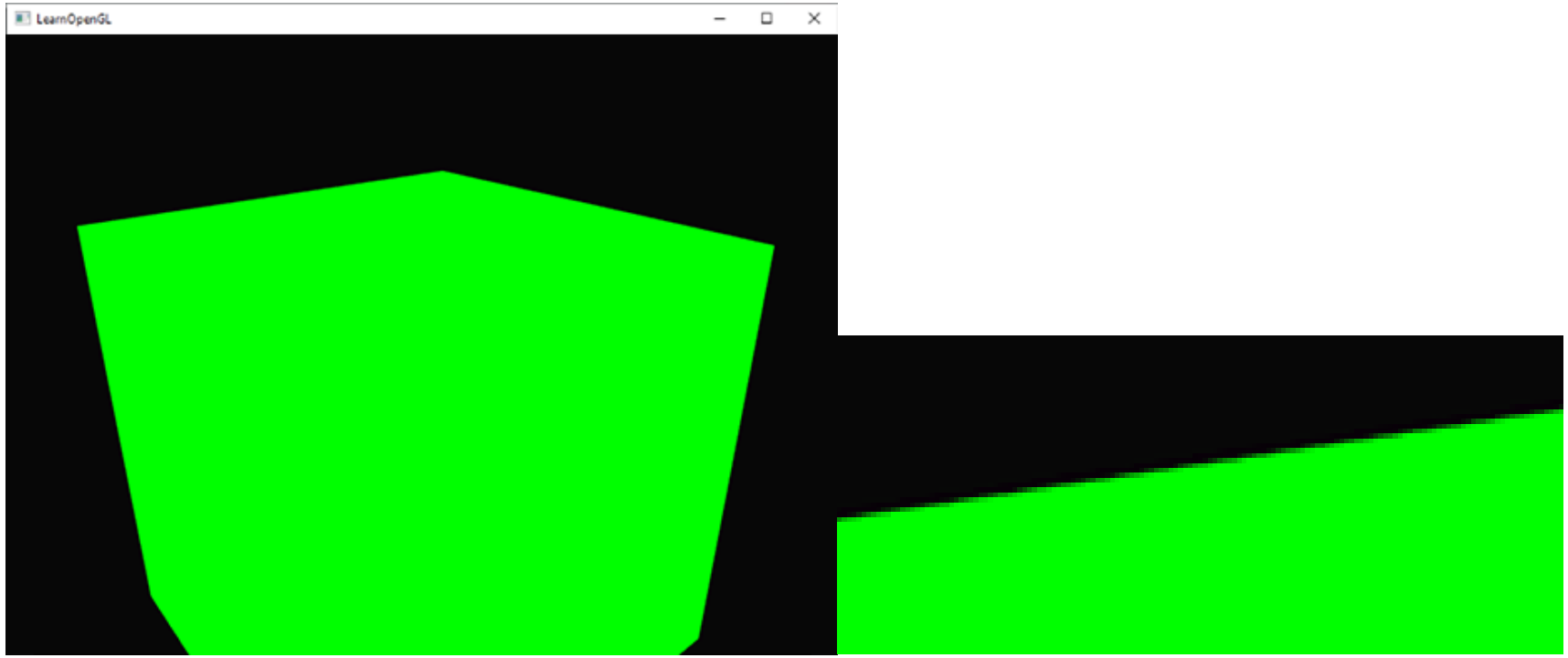
- MSAA는 가장 대표적인 anti-aliasing 방법임





MSAA in OpenGL

- 대부분의 윈도우 시스템은 multisample 버퍼를 제공
  - glfw 사용시 윈도우 생성 전 subsample의 개수를 미리 지정 `glfwWindowHint(GLFW_SAMPLES, 4);`
- 이후 OpenGL에서 GL_MULTISAMPLE을 활성시켜 주면 끝 `glEnable(GL_MULTISAMPLE);`
  - 대부분의 OpenGL 드라이버에서 기본적으로 활성화되어 있으나, 혹시 모르니 enable하도록 함
- 렌더링 결과



Off-screen MSAA

- 별도의 FBO를 만들어 사용시에는 multisample buffer도 직접 만들어야 함
- Multisampled texture attachments
  - glTexImage2DMultisample() 함수의 맨 마지막 파라미터는 fixedsamplelocations로, GL_TRUE로 설정

```
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, tex);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, samples, GL_RGB, width, height, GL_TRUE);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);
```

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D_MULTISAMPLE, tex, 0);
```

- Multisampled renderbuffer objects

```
glRenderbufferStorageMultisample(GL_RENDERBUFFER, 4, GL_DEPTH24_STENCIL8, width, height);
```

Off-screen MSAA

- Render to multisampled framebuffer
  - 이러한 멀티샘플된 버퍼의 이미지는 기존 프레임버퍼의 이미지처럼 shader에서 sampler2D 형태로 직접 샘플링하여 사용하는 것이 불가능
  - glBlitFramebuffer() 함수를 이용한 별도의 Resolve 단계를 통해, 원래 크기의 다른 프레임버퍼로 downscale하여 복사(blit)를 해 줘야 함

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, multisampledFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

Off-screen MSAA

- Post-processing에 MSAA 적용
  - Post-processing 효과를 위해 멀티샘플된 버퍼의 이미지를 텍스처로 사용하고 싶을 때에는, 이를 post-processing을 위한 별도의 FBO에 resolve해 준 다음에 사용해야 함

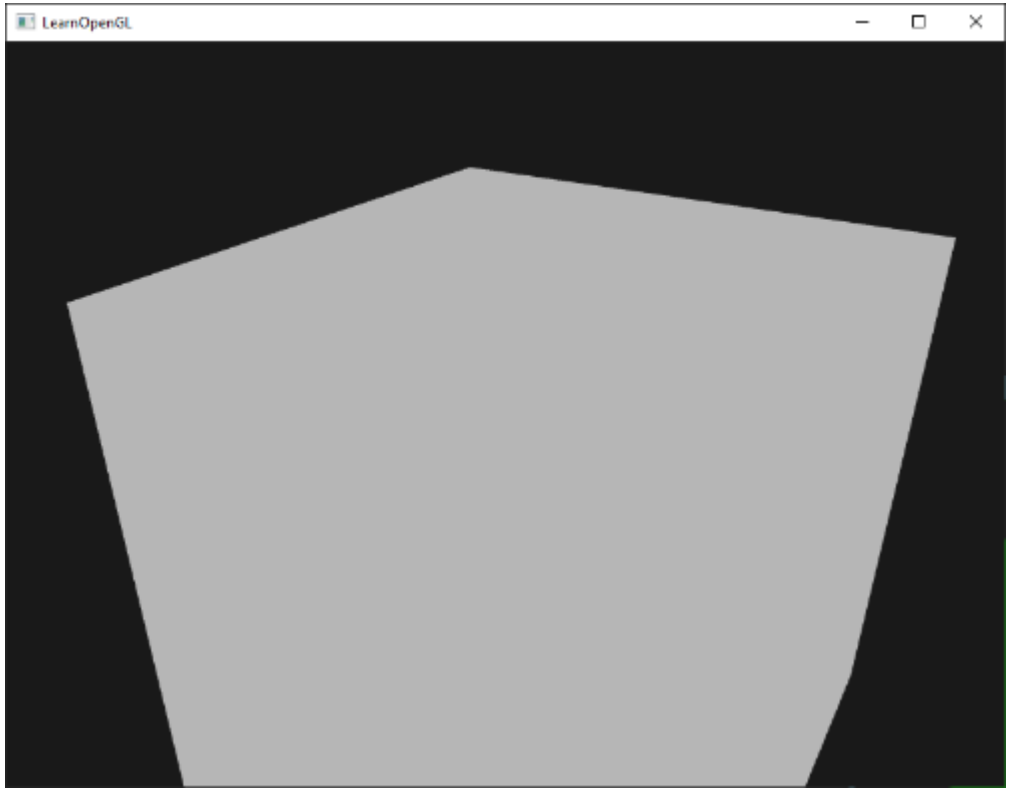
```
unsigned int msFBO = CreateFBOWithMultiSampledAttachments();
// then create another FBO with a normal texture color attachment
[...]
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, screenTexture, 0);
[...]
while(!glfwWindowShouldClose(window))
{
    [...]

    glBindFramebuffer(msFBO);
    ClearFrameBuffer();
    DrawScene();
    // now resolve multisampled buffer(s) into intermediate FBO
    glBindFramebuffer(GL_READ_FRAMEBUFFER, msFBO);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, intermediateFBO);
    glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);
    // now scene is stored as 2D texture image, so use that image for post-processing
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    ClearFramebuffer();
    glBindTexture(GL_TEXTURE_2D, screenTexture);
    DrawPostProcessingQuad();

    [...]
}
```

Off-screen MSAA

- Post-processing에 MSAA 적용 (cont.)
  - 오른쪽과 같은 grayscale filter 사용시에도 부드러운 edge를 확인 가능
  - 단, edge-detection filter와 같은 효과를 적용하면 다시 aliasing이 발생 가능



Custom Anti-Aliasing Algorithm

- 멀티 샘플링된 텍스처 이미지를 shader에 직접 전달하여 샘플링하는 방법
  - sampler2D 대신 sample2DMS 형태로 전달

```
uniform sampler2DMS screenTextureMS;
```

  - texture() 함수 대신 texelFetch() 함수를 사용하여 fetch할 subsample을 직접 지정

```
vec4 colorSample = texelFetch(screenTextureMS, TexCoords, 3);  // 4th subsample
```

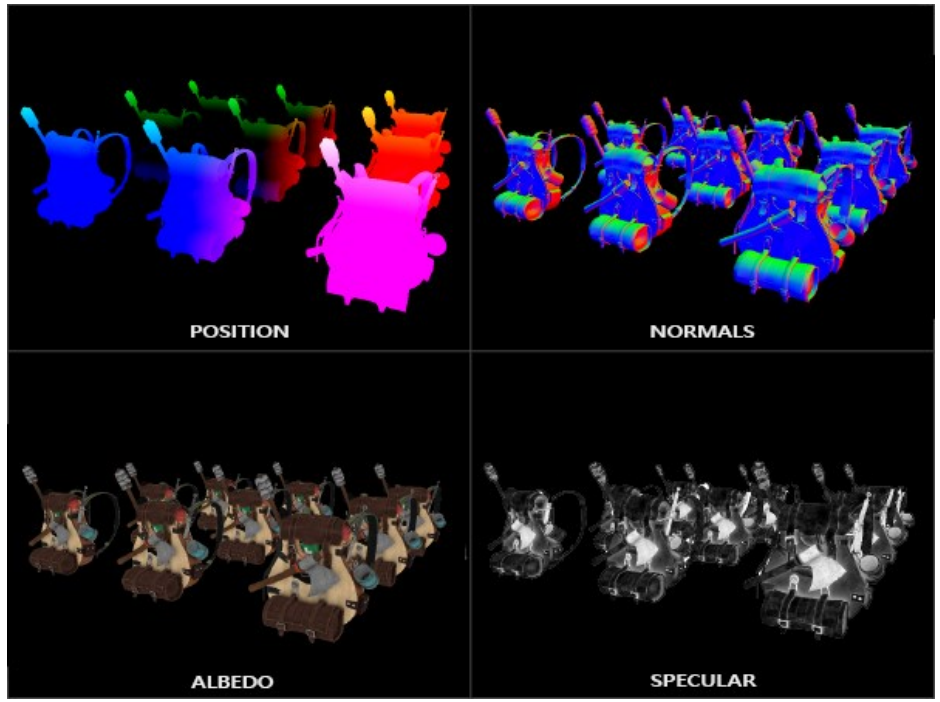
- 위 기능을 이용하여 custom anti-aliasing 알고리즘 구현 가능
  - 뒤에 소개할 SMAA S2x가 그 예

MSAA의 단점

- 메모리 사용량 & 액세스 증가
  - 15주차에 소개할 deferred shading & G-buffer 사용시 어마무시한 메모리 사용
  - 예) 4K 해상도에서, 4개 32bit render target으로 구성된 G-buffer를, 8x MSAA로 구성할 경우

    width * height * bytes per render target * # of render targets * # of subsamples

    = 4096 * 2160 * 4 * 4 * 8 = 1.08GB (!)

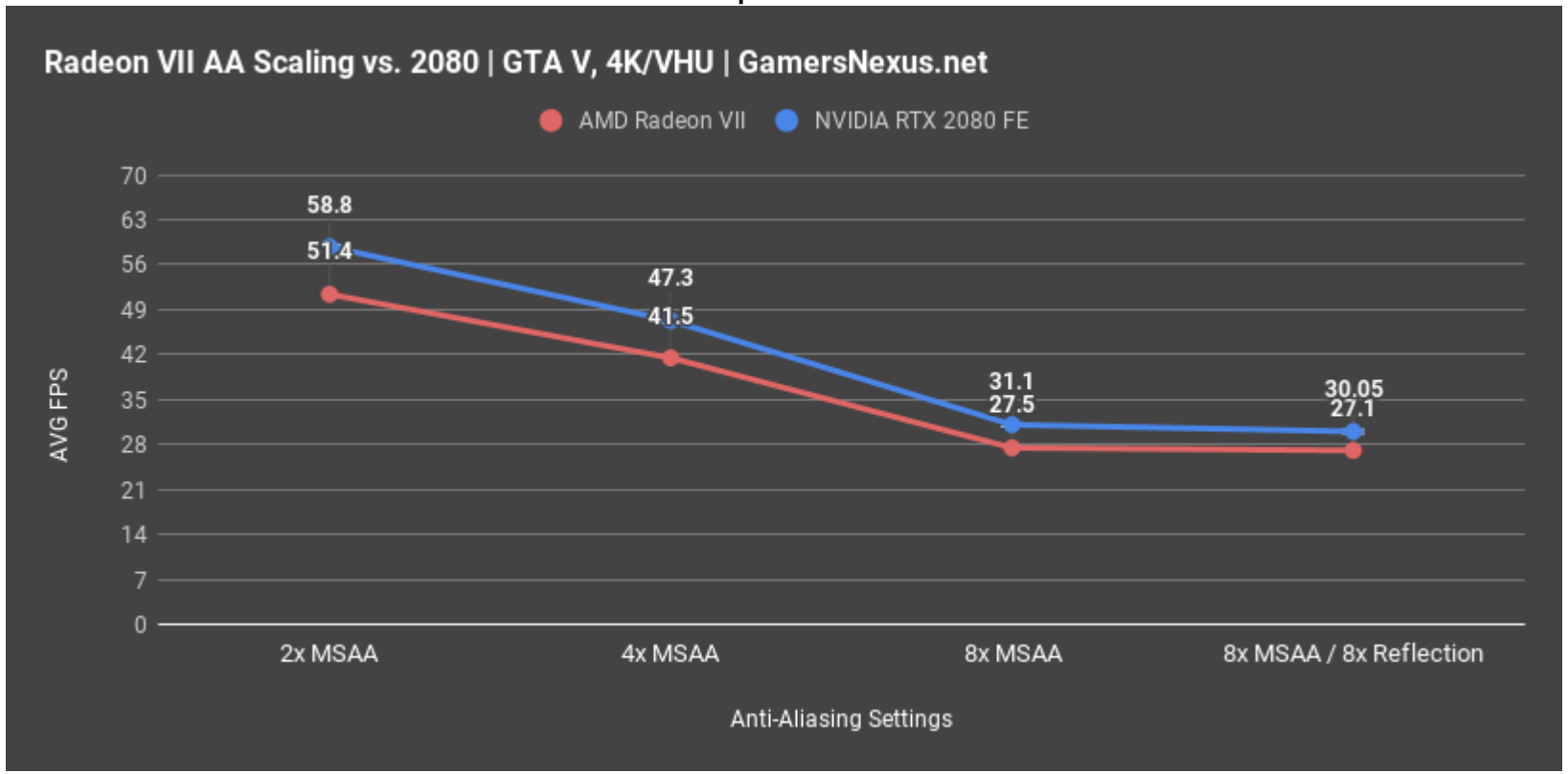


G-buffer

Deferred shading 결과

MSAA의 단점

- 성능 저하
  - 높아진 메모리 액세스량 및 custom resolve 단계로 인해 성능 저하 발생 가능
  - 렌더링 방식, GPU 구조, subsample의 개수, 해상도에 따라 성능 저하폭은 크게 달라질 수 있음



AMD Radeon VII Anti-Aliasing & 5K, 8K Benchmarks vs. RTX 2080 | GamersNexus - Gaming PC Builds & Hardware Benchmarks

MSAA의 단점

- 충분한 AA 품질을 위해서는 subsample 개수 증가가 필수
  - 더 많은 subsample = 성능 저하

- Geometry의 edge만 AA 가능
  - Texture 내부의 aliasing이나,
    post-processing으로 인해 생긴 aliasing은 처리 불가능

A Quick Overview of MSAA – The Danger Zone (wordpress.com)

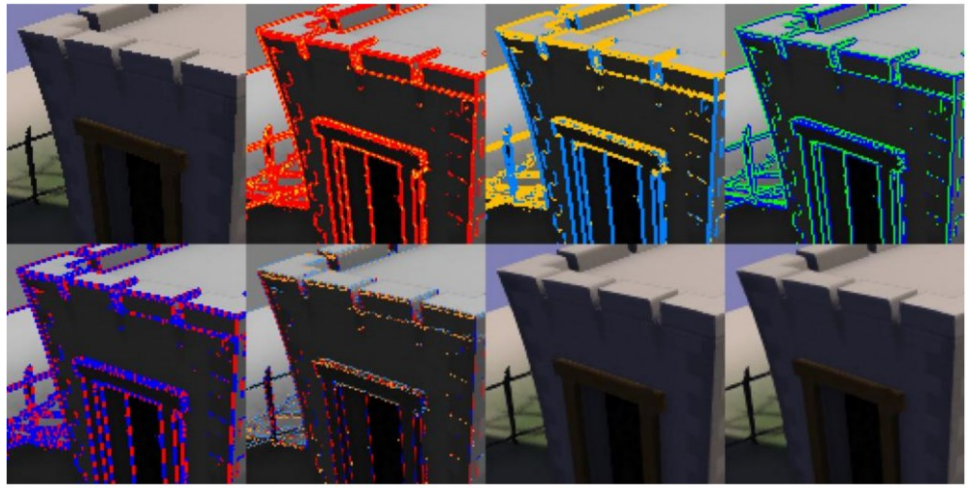
MSAA의 대안 – Post-Processing Anti-Aliasing

- Post-Processing Anti-Aliasing (PPAA)
  - 이미지 프로세싱에서 많이 사용되는 edge filtering 기법을 post-processing으로 구현한 방법들
  - 장점: MSAA와 달리 추가 메모리를 필요로 하지 않으며, 성능 하락폭도 상대적으로 적음
  - 단점: edge를 잘못 판별하여 불필요한 blur를 발생시킬 수 있고,
    subsample 부족으로 끊긴 edge를 filtering 못 할 수도 있음

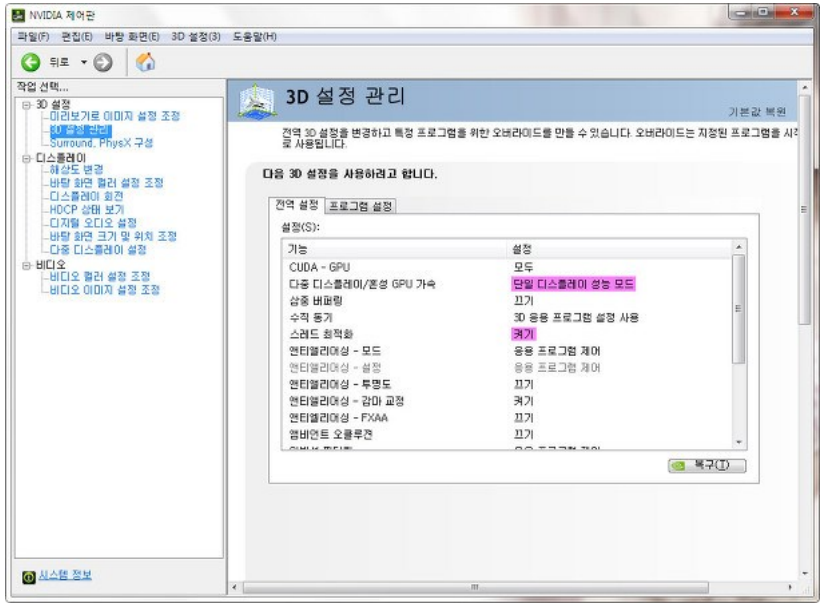
- FXAA, MLAA, CMAA, SMAA 등이 대표적
  - 대부분의 AAA 게임들은 이 중 일부를 구현하여, 사용자가 설정 가능하도록 기능 제공
  - GPU 드라이버 단에서 앱들에 강제 적용 가능하기도 함

FXAA (Fast approXimate Anti-Aliasing)

- 엔비디아에서 만든, Single-pass shader 상에서 edge를 판단하여 이를 부드럽게 (blur) 처리하는 방법
  - 속도가 빠르지만 edge가 아닌 부분(텍스처 내부나 font 등)도 흐려질 수 있음
- NVIDIA 제어판에서 '앤티앨리어싱 – FXAA' 로 일반 앱에 적용 가능

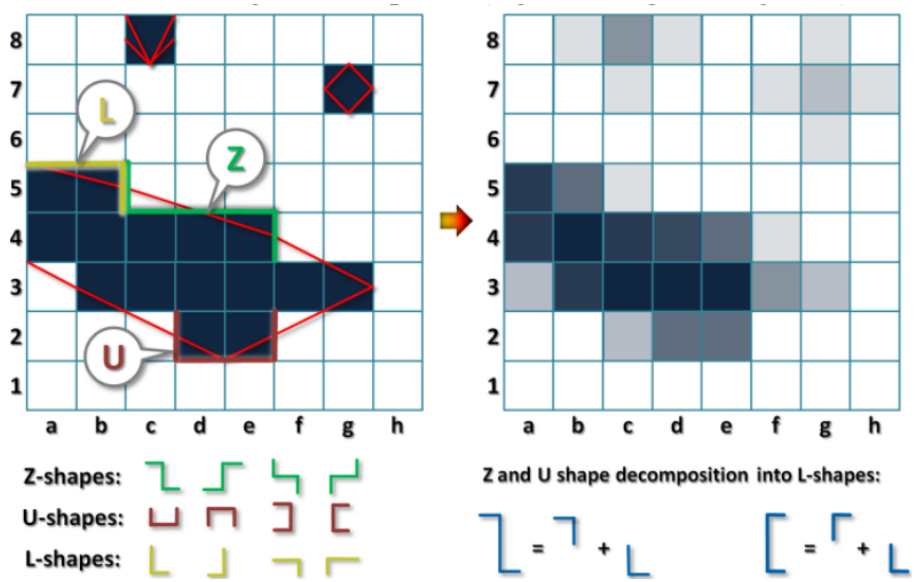


FXAA White paper (nvidia.com)

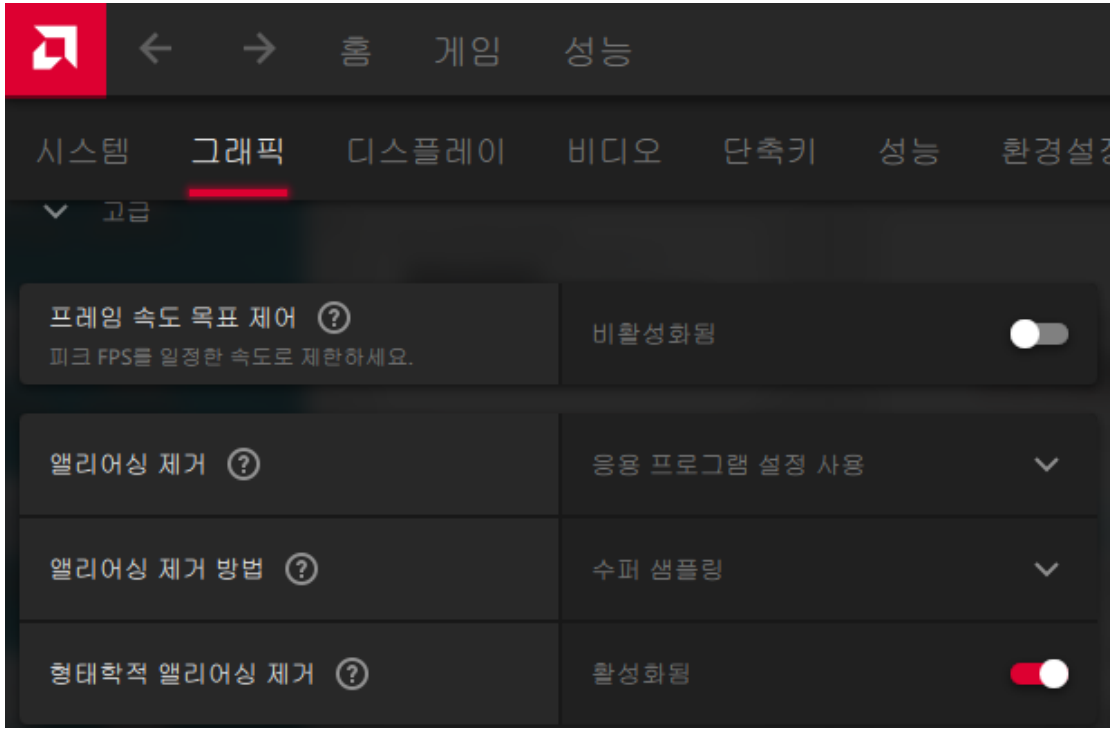


MLAA (Morphological Antialiasing)

- 인텔에서 HPG 2009 학회에서 소개한 방법
- 이미지를 형태학적으로 분석, edge를 Z/U/L-shapes로 분류하여 이를 각기 다르게 처리하는 방법
- AMD Radeon Software에서는 “형태학적 앨리어싱 제거”로 일반 앱에 적용 가능



Morphological antialiasing (acm.org)



CMAA (Conservative Morphological Anti-Aliasing)

- MLAA의 개선판
  - Symmetrical Z shapes 판별 – Long edge의 AA 품질 증가
  - Locally dominant edge 판별 – 불필요한 필터링 방지
  - CMAA2에서는 성능을 더욱 개선하고, MSAA와의 결합과 같은 기능 추가
- 인텔 그래픽 제어판에는 '일반 형태성 앤티앨리어싱'으로 기술

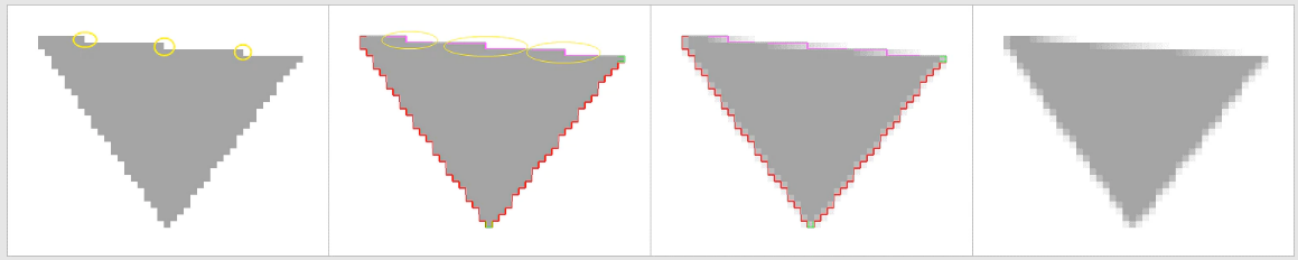


Figure 7. Typical detection and handling of symmetrical Z shapes (circled in yellow)

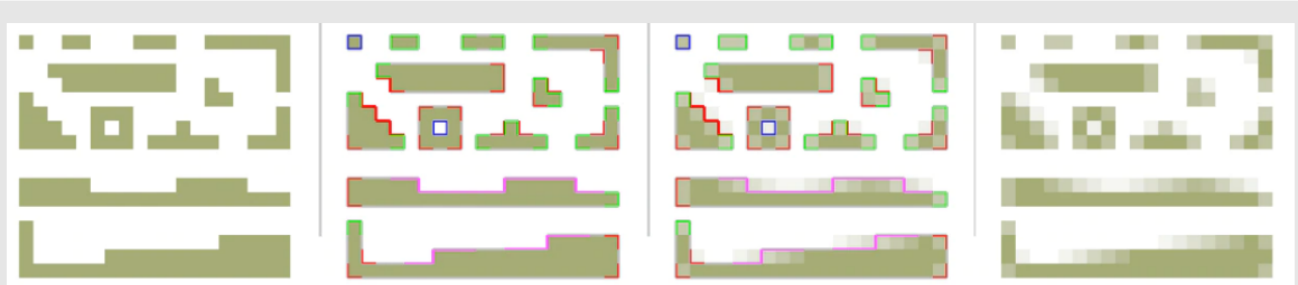
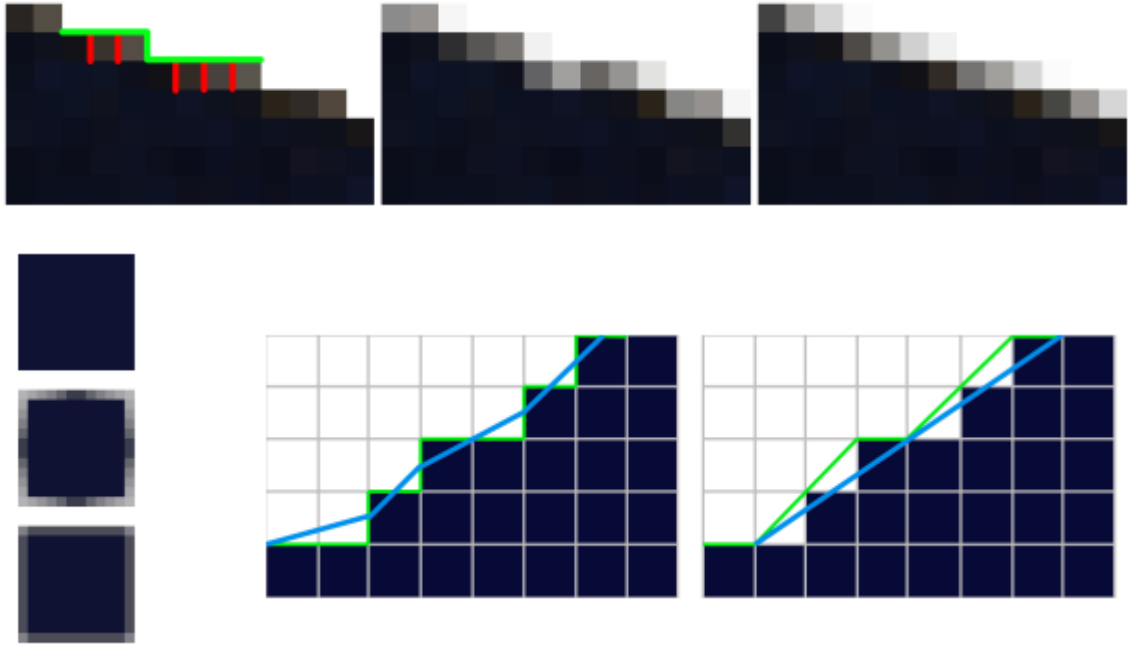


Figure 8. All CMAA shapes: original image, edge detection and final anti-aliased image (with/without edges)

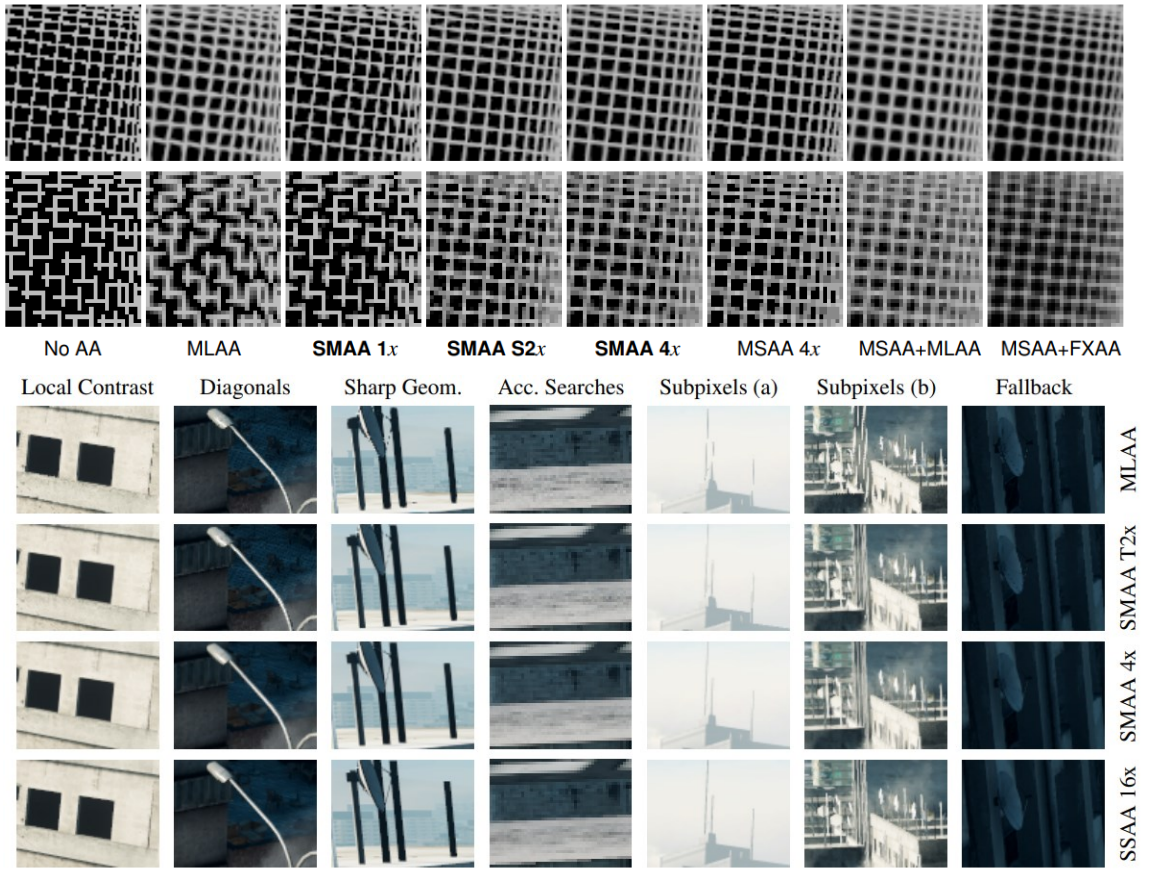
Conservative Morphological Anti-Aliasing (CMAA) (intel.com)

SMAA (Enhanced Subpixel Morphological Antialiasing)

- Universidad de Zaragoza와 Crytek이 함께 내 놓은, MLAA의 또 다른 개선판
  - Edge 판별, 대각선 처리, shape의 보존 등 전방위적으로 MLAA의 품질 개선 (그만큼 성능은 하락)
  - Temporal super-sampling 및 MSAA와의 결합도 가능

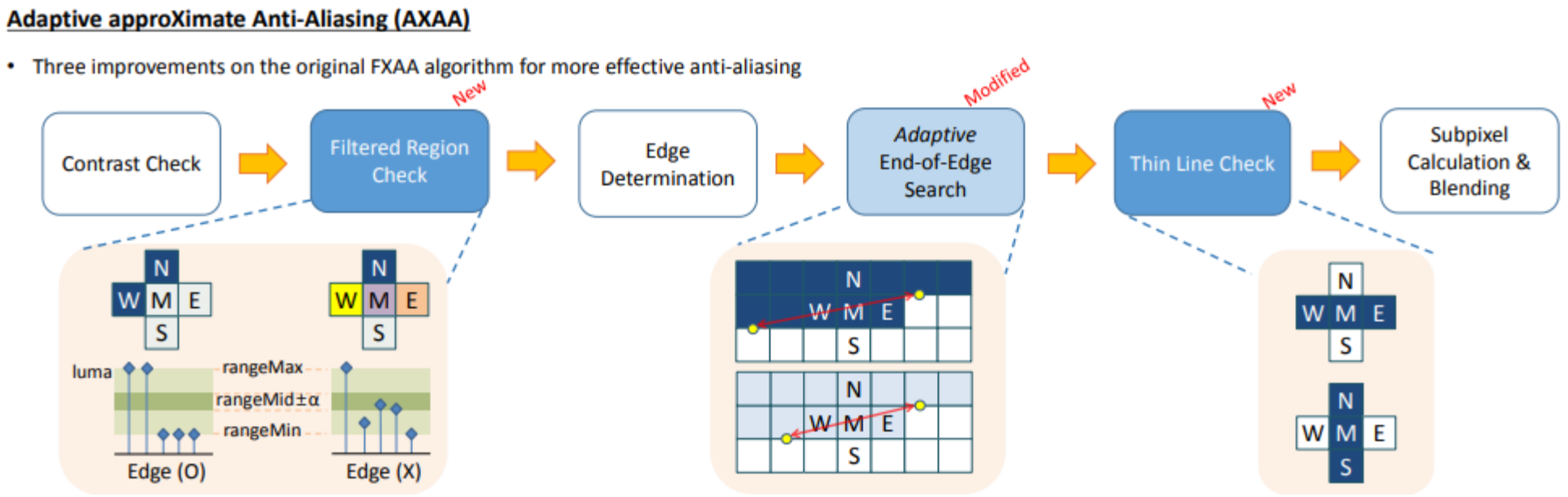


SMAA: Enhanced Subpixel Morphological Antialiasing (iryoku.com)



AXAA (Adaptive approXimate Anti-Aliasing)

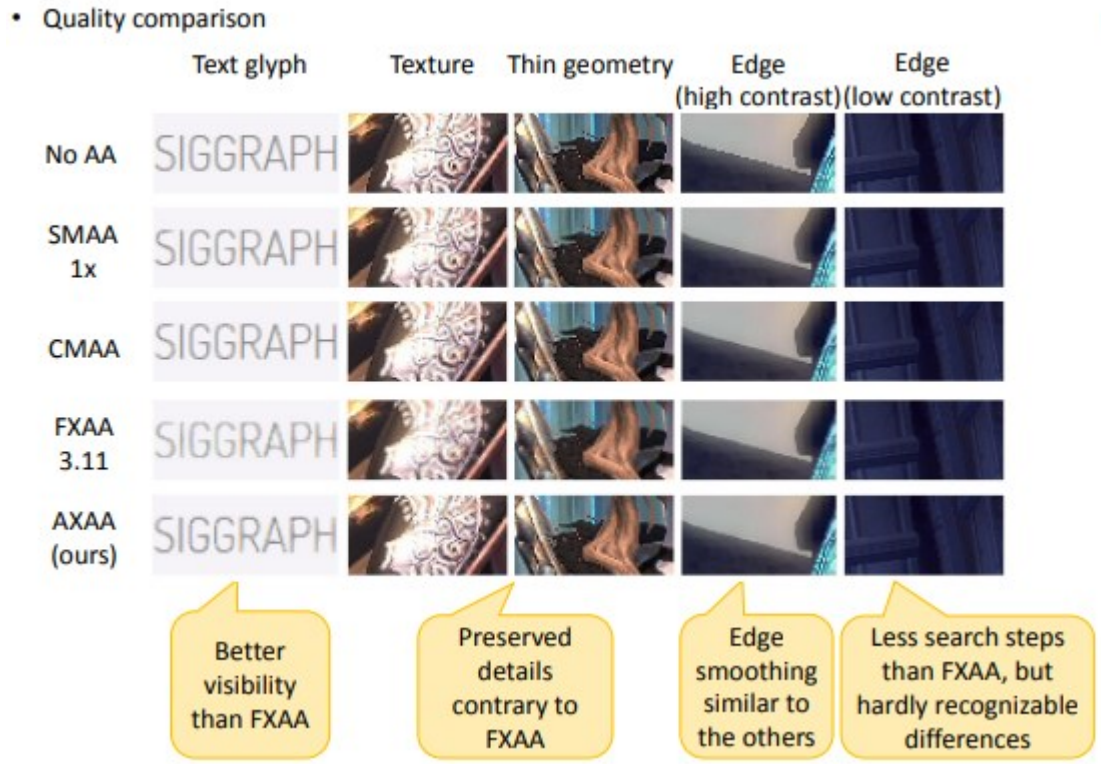
- Nah 등이 2016년 SIGGRAPH에서 발표한, FXAA의 개선판
  - 미리 필터링된 영역을 체크하여, 이 부분은 추가 필터링을 하지 않음으로써 텍스처의 흐려짐 형상 방지
  - Contrast에 따라 search 영역을 조정하여 성능 향상
  - 1픽셀 단위의 thin line을 필터링하지 않도록 하여 작은 font의 가독성 증가

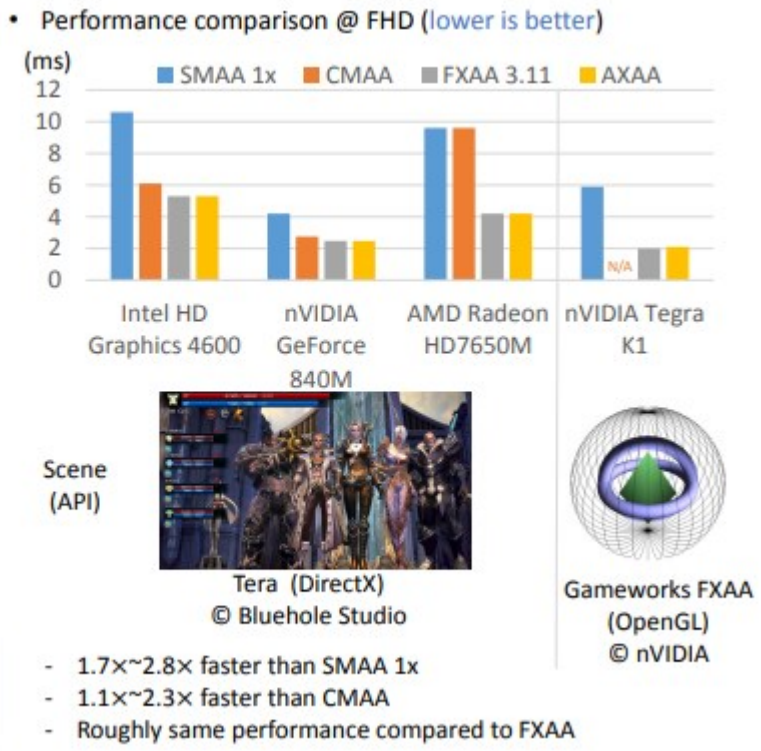


AXAA: Adaptive approXimate Anti-Aliasing (nahjaeho.github.io)

AXAA (Adaptive approXimate Anti-Aliasing)

- 성능 및 품질 비교
  - FXAA 대비 속도는 유사하지만 품질은 대폭 향상
  - CMAA 및 SMAA 1x 대비 품질은 유사하지만 속도는 최고 2.3~2.8배까지 향상
- 셰이더 코드 공개 https://nahjaeho.github.io/papers/SIG2016/Fxaa3_11.h

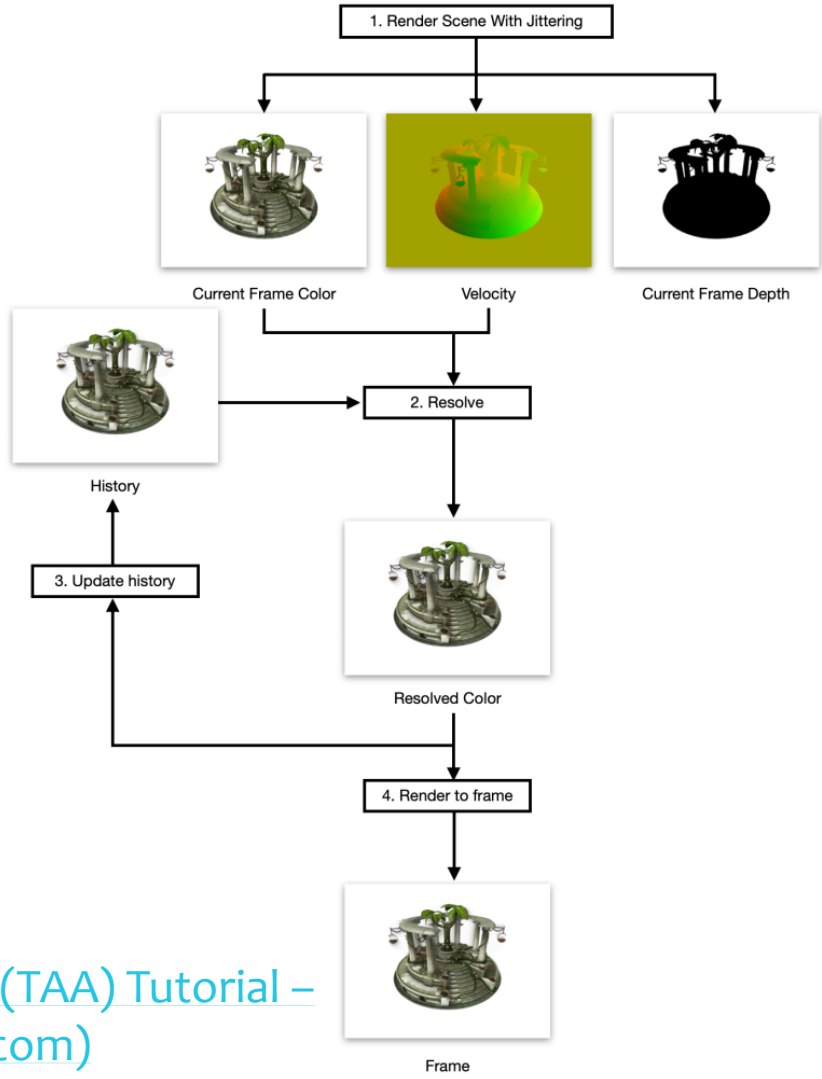




MSAA의 또 다른 대안 - TAA (Temporal Anti-Aliasing)

- 이전 프레임에서 sample을 취하여 anti-aliasing을 하는 방법
  - 현재 프레임의 픽셀과 이전 프레임의 픽셀을 match시켜야 함
  - Velocity map 및 이전 프레임의 history color buffer를 이용하여 이전 프레임의 픽셀을 현재 프레임의 픽셀과 섞음
- 장점: 시점이 바뀔 때 눈에 띄는 aliasing을 효과적으로 제거
- 단점: 잘못된 sample을 취할 경우 ghosting 현상 발생 가능
  - 빠른 속도로 움직이는 물체에서 주로 나타남

트윗 / 트위터 (twitter.com)



Temporal Anti-Aliasing(TAA) Tutorial – Sugu Lee (wordpress.com)

TAA vs MSAA

- Forza Horizon 5: TAA vs MSAA vs OFF | 1440p, RTX 2060 - YouTube

© nvidia

마무리

마무리

- Advanced OpenGL의 마지막 시간으로, 아래와 같은 내용을 살펴보았습니다.
  - Instancing
  - Anti Aliasing

- 이어서 Advanced Lighting (1) 이론 수업이 진행됩니다.

- 다음 실습 시간에는 아래 실습을 수행할 예정입니다.
  - 기반 코드(LearnOpenGL 4.10.2)를 instancing을 사용하도록(4.10.3) 변경
  - 여기에 샘플 개수를 달리하여 MSAA의 효과 측정
  - FPS로 성능을 측정할 예정이니 Fraps(윈도우) 또는 Count It(Mac)을 깔아서 오세요!
    Download Fraps 3.5.99 free version
    Count It (Mac Frame Rate counter): 2020 Update | Mac Gamer HQ