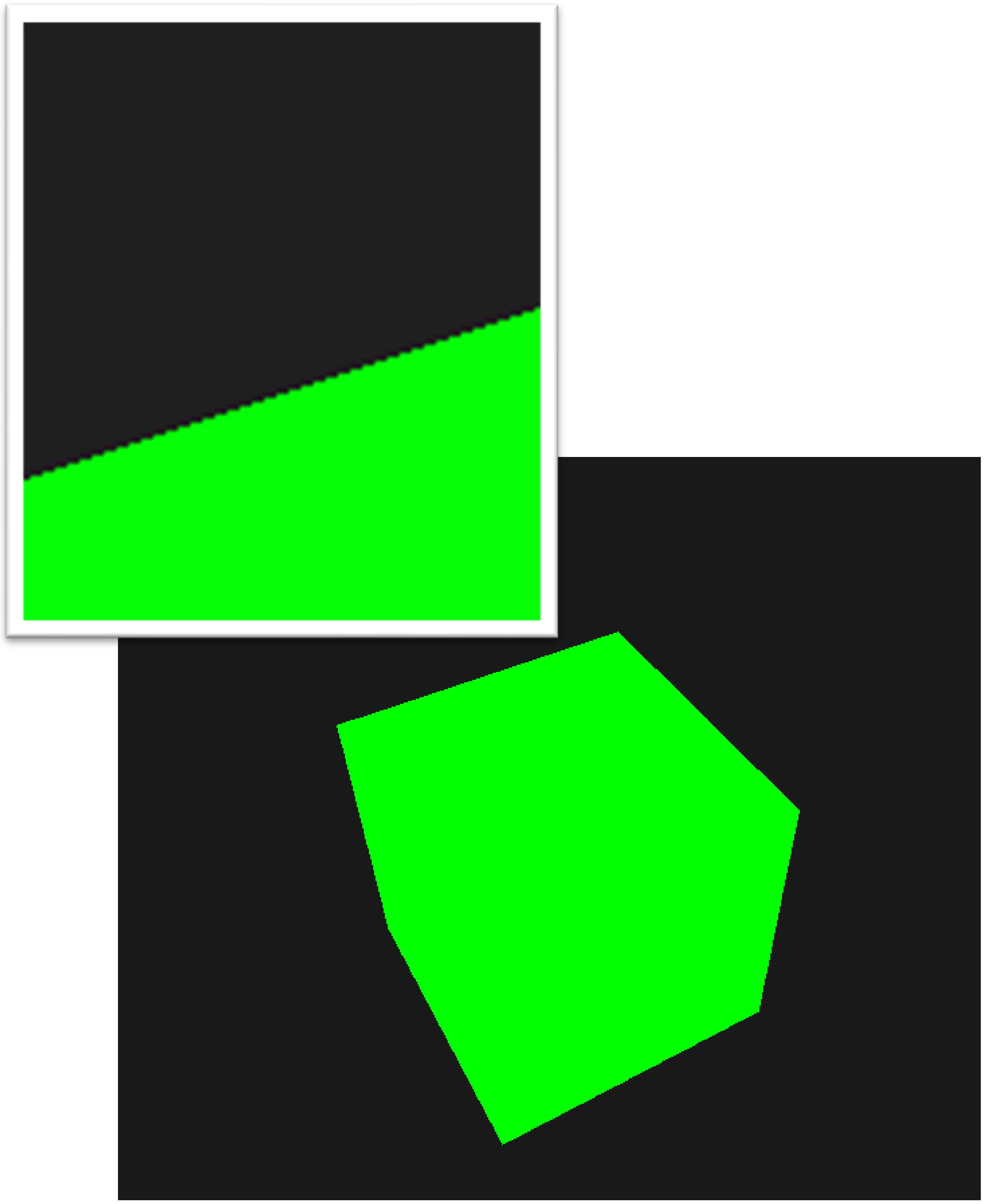
Computer Graphics II – Anti Aliasing

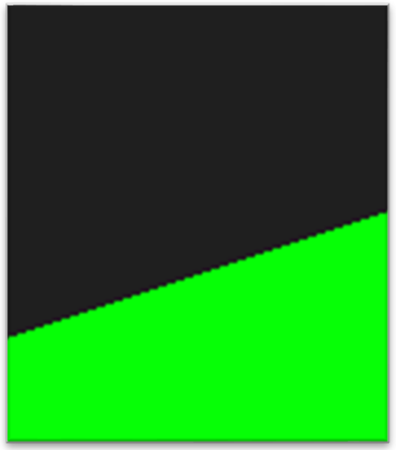
Kai Lawonn

Introduction

- You probably came across some jagged saw-like patterns along the edges
- Reason is how the rasterizer transforms the vertex data into fragments
- E.g., drawing a simple cube:



Introduction



- Do not want in a final version of an application
- This effect is called aliasing
- There are quite a few techniques out there called anti-aliasing techniques that fight exactly this aliasing behavior to produce more smooth edges

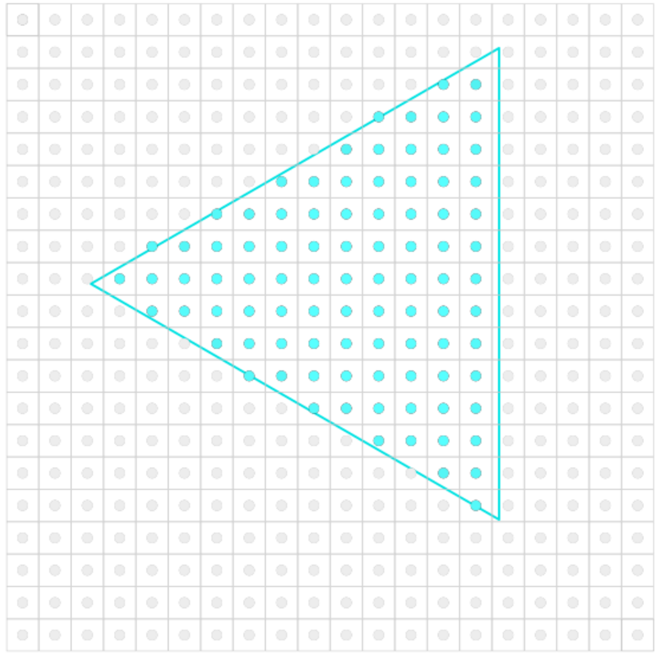
Introduction

- First technique: super sample anti-aliasing (SSAA) that temporarily; used higher resolution to render, the visual output is updated in the framebuffer, the resolution was downsampled back to the normal resolution
- Solution to the aliasing problem, but it came with a major performance drawback since we had to draw a lot more fragments than usual
- This technique therefore only had a short glory moment
- Then, multisample anti-aliasing or MSAA that borrows from the concepts behind SSAA while implementing a much more efficient approach
- We will discusse this MSAA technique that is built-in in OpenGL

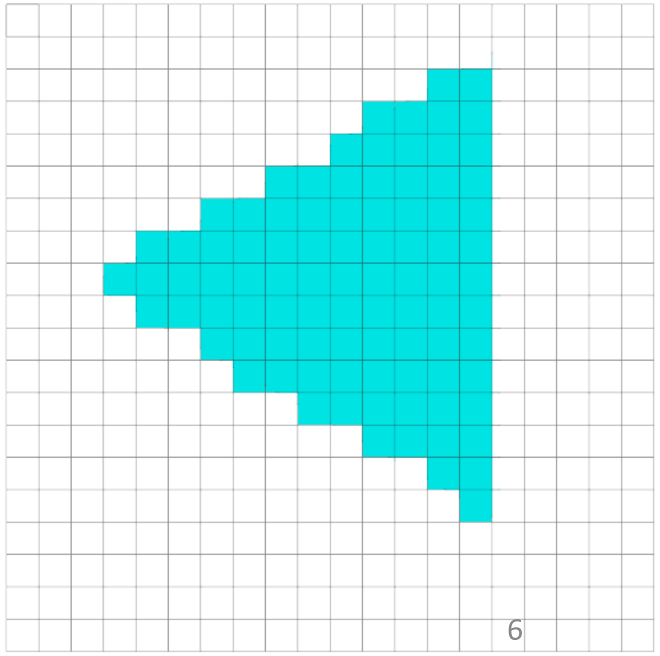
Multisampling

- To understand multisampling, we need to delve a bit further into the inner workings of OpenGL's rasterizer
- Rasterizer is the combination of all algorithms and processes that sit between your final processed vertices and the fragment shader
- It takes all vertices belonging to a single primitive and transforms this to a set of fragments
- Vertex coordinates can theoretically have any coordinate, but fragments can't since they are bound by the resolution of your window
- There will almost never be a one-on-one mapping between vertex coordinates and fragments, so the rasterizer has to determine in some way at what fragment/screen-coordinate each specific vertex will end up at

Multisampling



- A grid of screen pixels, center contains a sample point to determine if a pixel is covered by the triangle
- The cyan sample points are covered by the triangle and a fragment will be generated for that covered pixel
- Some parts of the triangle edges enter screen pixels, pixel's sample point is not covered by the triangle, so this pixel won't be influenced by any fragment shader
- Origin of aliasing - rendered version of the triangle would look like this on your screen:

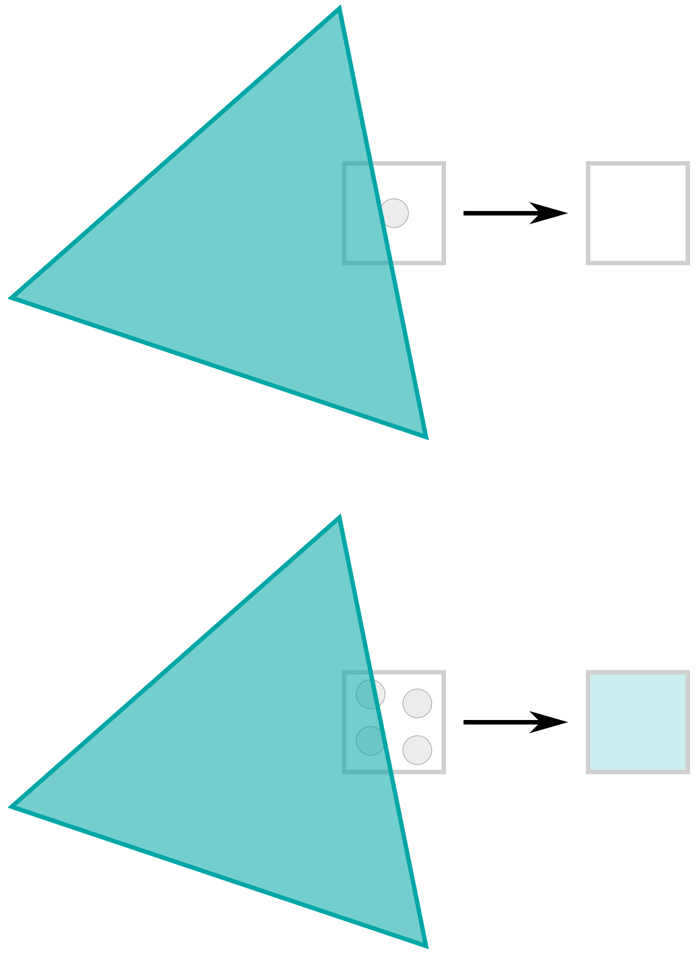


Multisampling

- Limited amount of screen pixels, some will be rendered along an edge
- Results in primitives with non-smooth edges
- Multisampling use multiple sample points

Multisampling

- Instead of a single sample point, place 4 subsamples
- Size of color buffer increased by the number of subsamples
- Top: normally determine the coverage of a triangle (pixel won't run fragment shader → remains blank)
- Bottom: multisampled, each pixel contains 4 sample points (only 2 cover the triangle)

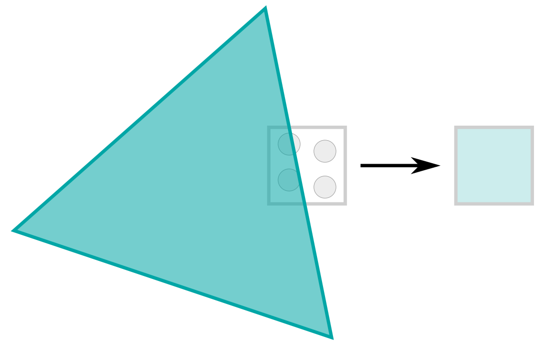


Multisampling

The amount of sample points can be any number

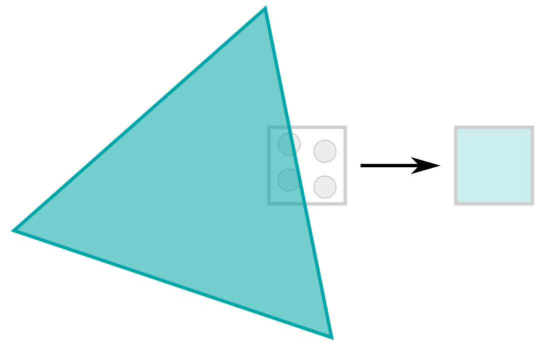
More samples giving us better coverage precision

Multisampling



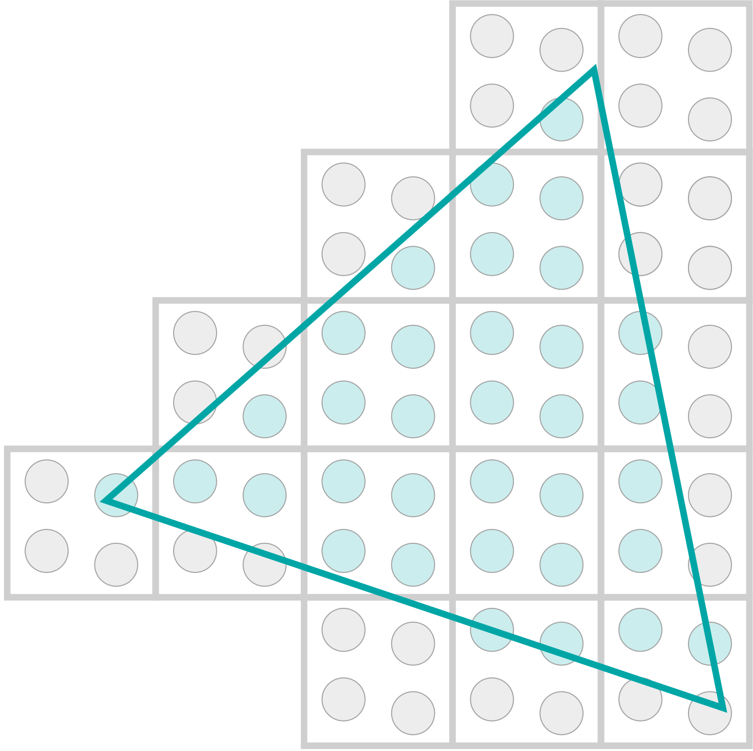
- 2 subsamples covered by the triangle, next determine a color
- Initial guess: run the fragment shader for each covered subsample and later average the colors of each subsample per pixel
- This case, fragment shader run twice on interpolated vertex data at each subsample and store the resulting color in those sample points
- Does not work this way, because had to run a lot more fragment shaders than without multisampling → reducing performance

Multisampling



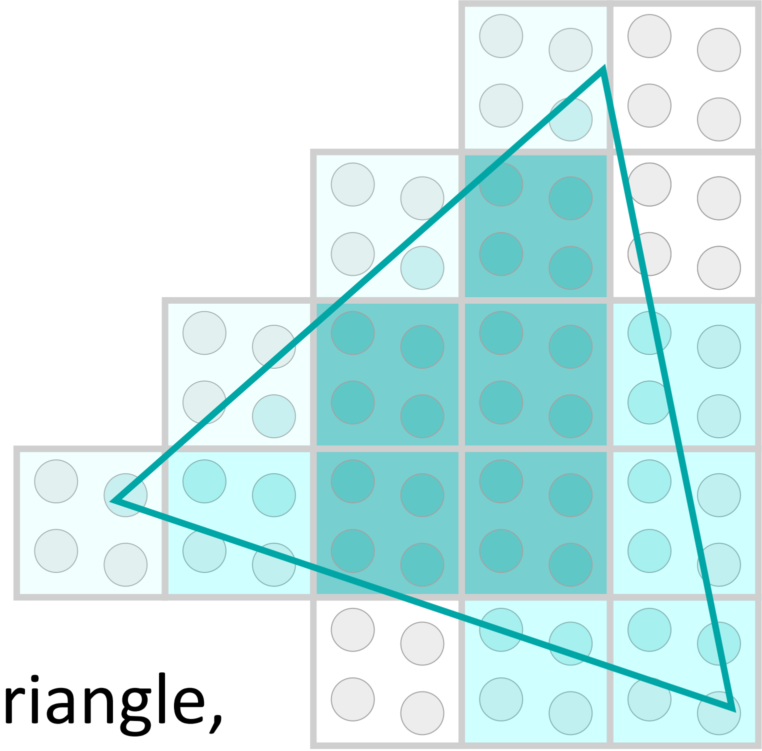
- MSAA: fragment shader is run once per pixel (for each primitive) regardless of how many subsamples the triangle covers
- It is run with the vertex data interpolated to the center of the pixel and the resulting color is then stored inside each of the covered subsamples
- Once the color buffer's subsamples are filled (with colors of the primitives), these colors are then averaged per pixel resulting in a single color per pixel
- Only 2 of 4 samples were covered, the color of the pixel was averaged with the triangle's color and the color stored at the other 2 sample points (in this case: the clear color)

Multisampling



- Result: color buffer where all the primitive edges now produce a smoother pattern
- Each pixel, 4 subsamples (blue subsamples are covered by the triangle and the gray sample points not)
- Within the inner region of the triangle all pixels will run the fragment shader once where its color output it is stored in all 4 subsamples
- At the edges not all subsamples will be covered so the result of the fragment shader is only stored at some subsamples
- Based on the amount of subsamples covered, the resulting pixel color is determined by the triangle color and the other subsample's stored colors

Multisampling



- The more sample points are covered by the triangle, the more the pixel color is that of the triangle:
- For each pixel, the less subsamples are part of the triangle, the less it takes the color of the triangle
- Edges surrounded by colors slightly lighter than the actual edge color → causes the edge to appear smooth when viewed from a distance

Multisampling

- Not only color, but also the depth and stencil test now make use of the multiple sample points
- Depth testing: the vertex’s depth interpolated to each subsample (before the depth test)
- Stencil testing: store stencil values per subsample, instead of per pixel
- This does mean that the size of the depth and stencil buffer are now also increased by the amount of subsamples per pixel
- This was just a basic overview of how multisampled anti-aliasing works
- Actual logic behind the rasterizer is a bit more complicated

MSAA in OpenGL

Introduction

- MSAA in OpenGL: need a color buffer that is able to store more than one color value per pixel (multisampling requires color per sample)
- Need a new type of buffer that can store a given amount of multisamples and this is called a multisample buffer

MSAA in OpenGL

- In GLFW a multisample buffer with N samples instead of a normal color buffer by calling glfwWindowHint before creating the window:

```
glfwWindowHint(GLFW_SAMPLES, 4);
```

MSAA in OpenGL

- With glfwCreateWindow the rendering window is created, with a color buffer containing 4 subsamples per screen coordinate
- GLFW also automatically creates a depth and stencil buffer with 4 subsamples per pixel
- This does mean that the size of all the buffers is increased by 4

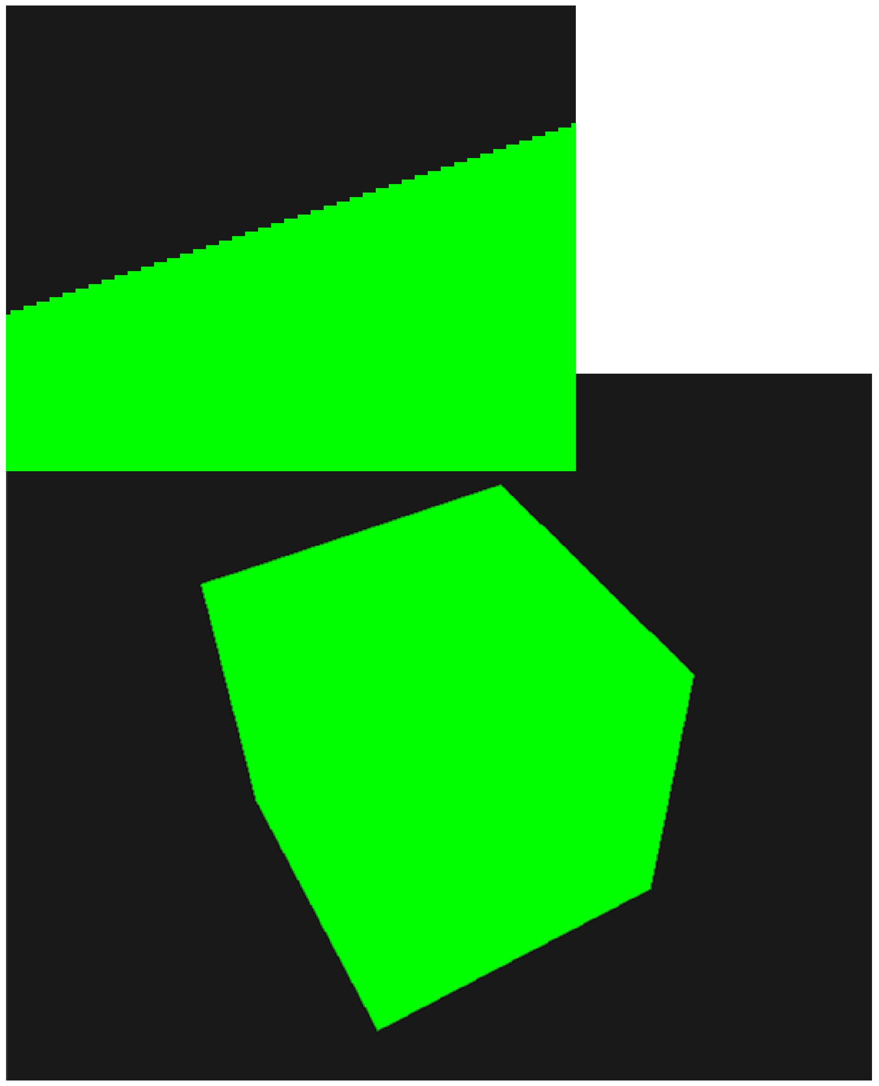
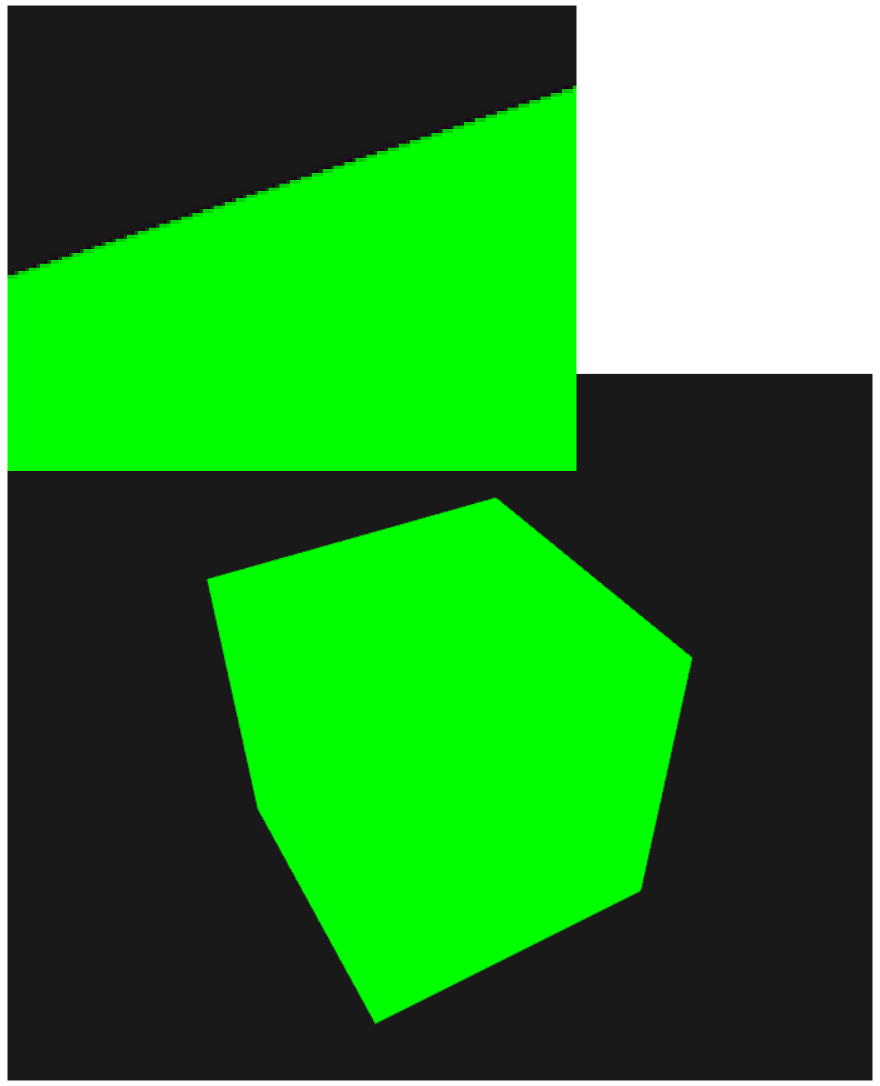
MSAA in OpenGL

- Need to enable multisampling by calling glEnable and enabling GL_MULTISAMPLE
- Mostly, multisampling is enabled by default so this call is redundant, but it's usually a good idea to enable it anyways
- This way all OpenGL implementations have multisampling enabled:

```
glEnable(GL_MULTISAMPLE);
```

F5…

- … smooth!

Off–screen MSAA

Introduction

- GLFW takes care of creating the multisampled buffers, enabling MSAA is quite easy
- If we want to use our own framebuffers however, for some off-screen rendering, have to generate the multisampled buffers ourselves → take care of creating multisampled buffers
- Two ways we can create multisampled buffers to act as attachments for framebuffers: texture attachments and renderbuffer attachments

Multisampled Texture Attachments

- To create a texture with multiple sample points, use glTexImage2DMultisample instead of glTexImage2D with GL_TEXTURE_2D_MULTISAPLE as its texture target:

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

- Second argument number of samples
- If the last argument is equal to GL_TRUE the image will use identical sample locations and the same number of subsamples for each texel

Multisampled Texture Attachments

- To attach a multisampled texture to a framebuffer: we use glFramebufferTexture2D with GL_TEXTURE_2D_MULTISAMPLE:

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

- The currently bound framebuffer now has a multisampled color buffer in the form of a texture image

Multisampled Renderbuffer Objects

- Creating a multisampled renderbuffer object is like creating textures
- Change the call to glRenderbufferStorage that is now glRenderbufferStorageMultisample:

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

- Here, an extra parameter after the renderbuffer target → set the samples (4 in this particular case)

Render to Multisampled Framebuffer

- Rendering to a multisampled framebuffer object goes automatically
- Whenever we draw anything while the framebuffer object is bound, the rasterizer will take care of all the multisample operations
- Then end up with a multisampled color and/or depth and stencil buffer
- Multisampled buffer is a bit special → cannot directly use their buffer images for other operations like sampling them in a shader
- A multisampled image contains much more information than a normal image so what we need to do is downscale or resolve the image
- Resolving a multisampled framebuffer is generally done via glBlitFramebuffer that copies a region from one framebuffer to the other while also resolving any multisampled buffers

Render to Multisampled Framebuffer

- glBlitFramebuffer transfers a source region (by 4 screen-space coordinates) to a given target region (by 4 screen-space coordinates)
- If bind to GL_FRAMEBUFFER we're binding to both the read and draw framebuffer targets, could also bind targets individually with GL_READ_FRAMEBUFFER and GL_DRAW_FRAMEBUFFER

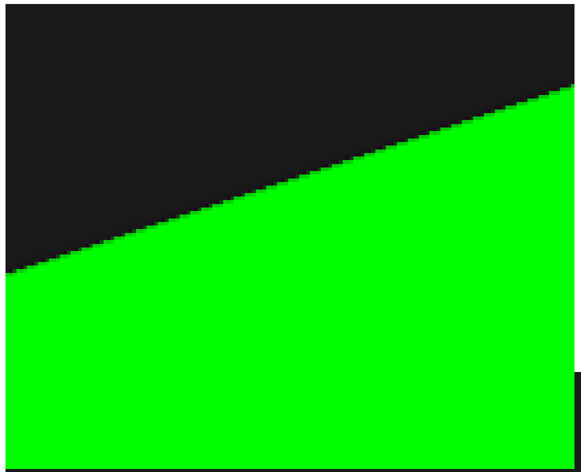
Render to Multisampled Framebuffer

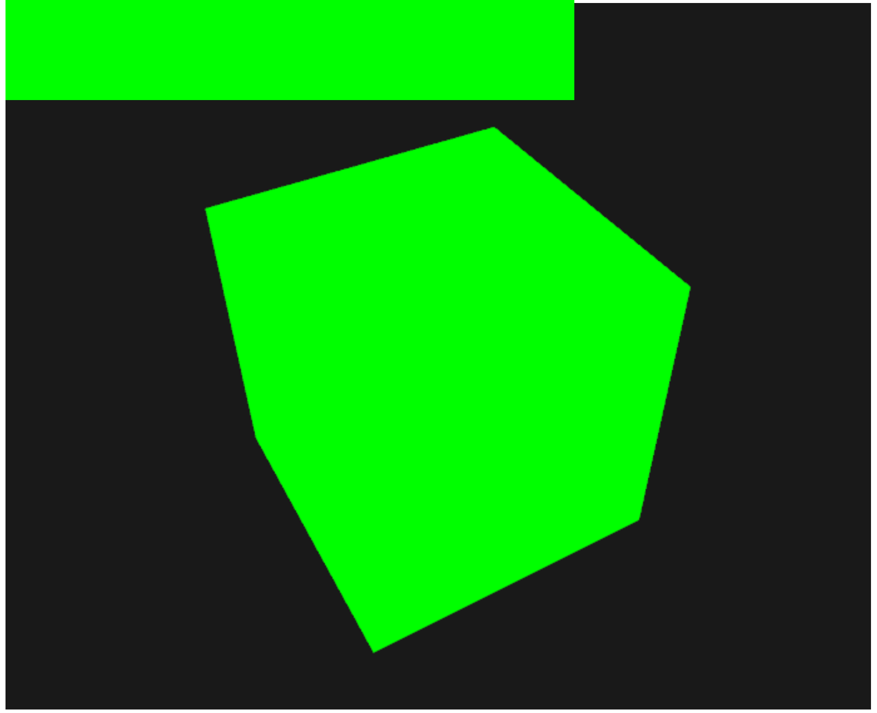
- The glBlitFramebuffer function reads from those two targets to determine which is the source and which is the target framebuffer
- We could then transfer the multisampled framebuffer output to the actual screen by blitting the image to the default framebuffer like so:

```
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);
```

F5…

- … same!





Render to Multisampled Framebuffer

- What if we want to do post-processing on the texture result of a multisampled framebuffer
- Cannot directly use the texture(s) in the fragment shader
- We could blit the multisampled buffer(s) to a different FBO with a non-multisampled texture attachment
- Then use this color attachment texture for post-processing
- Have to generate a new FBO that acts solely as an intermediate framebuffer object to resolve the multisampled buffer into a normal 2D texture we can use in the fragment shader

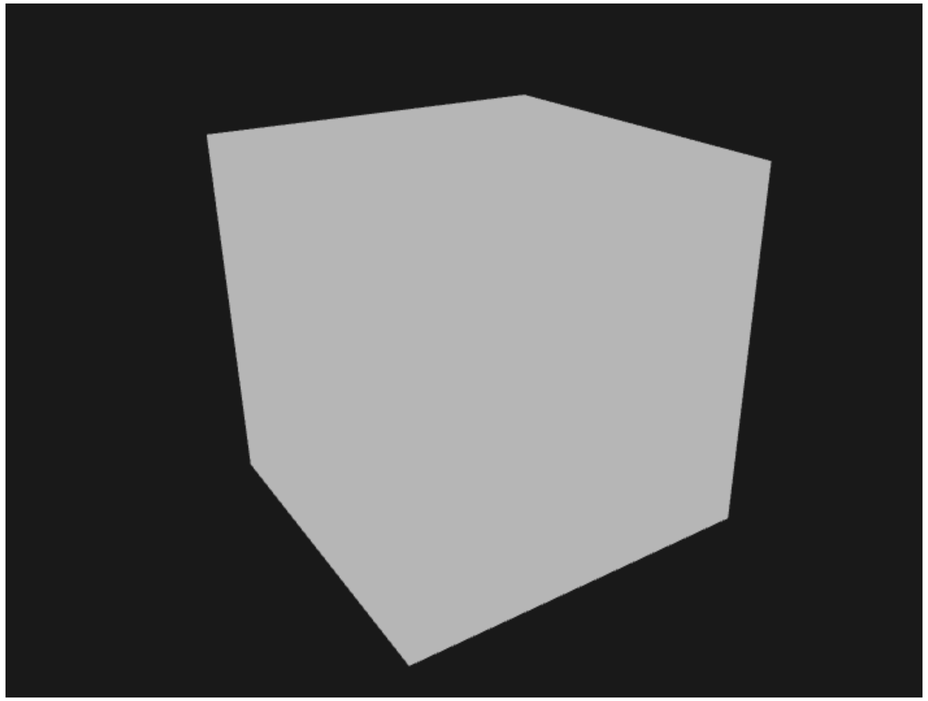
Render to Multisampled Framebuffer

- This process looks a bit like this in pseudocode:

```
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();
  ...
}
```

F5…

- … greyscale processing



Screen texture is a normal texture with just a single sample point → some filters, e.g., edge-detection lead to jagged edges again

To accommodate, could blur the texture afterwards or create own anti-aliasing algorithm

Notes

- Combine multisampling with off-screen rendering, to take care of some extra details
- All the details are worth the extra effort though since multisampling significantly boosts the visual quality of the scene
- Enabling multisampling can noticeably reduce performance
- As of this writing, using MSAA with 4 samples is commonly preferred

Custom Anti-Aliasing Algorithm

- Possible to directly pass a multisampled texture image to the shaders instead of first resolving them
- GLSL gives us the option to sample the texture images per subsample → create our own anti-aliasing algorithms

Custom Anti-Aliasing Algorithm

- To retrieve the color value per subsample, use sampler2DMS instead of the usual sampler2D:

```
uniform sampler2DMS screenTextureMS;
```

- Using the texelFetch function it is then possible to retrieve the color value per sample:

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

Questions???