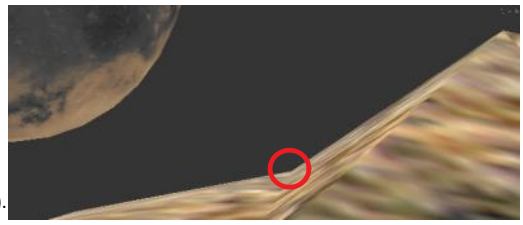
ANTI-ALIASING

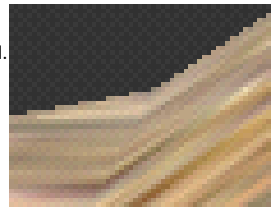


You may have noticed that the edges of your objects may sometimes appear jagged. If you look closely and the edges of the objects in the image to the right, you can see the edges are not smooth.

- This is due to how the rasterizer transforms the vertex data into actual fragments.
- The effect of clearly seeing the pixel formations an edge is composed of is called **aliasing**.
  - **Anti-aliasing** techniques exist to reduce aliasing, in an attempt to make edges look smoother.

The first commonly used technique was **Super Sample Anti-Aliasing (SSAA)**.

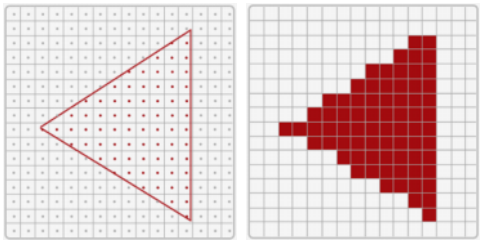
- This technique involves temporarily using a much higher resolution render buffer to render the scene in (super sampling). From there, the full scene is rendered and then downsampled back to the original resolution.
  - The extra resolution from the super sample was used to prevent these jagged edges.
  - This technique comes with a major performance draw back because of this, since we have to draw a significant amount more fragments than usual.
    - Because of this, this is more of a legacy technique.



Multisample Anti-Aliasing (MSAA) borrows some concepts from SSAA while implementing a more efficient approach.

Multisampling

To understand what multisampling is and how it works to solve aliasing, let's first take a look at how OpenGL's rasterizer works.

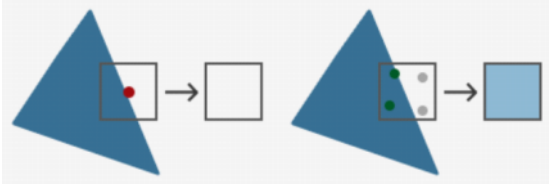


The rasterizer is the combination of all algorithms and processes that sit between your final processed vertices and the fragment shader.

- The rasterizer 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 screen.
- There will almost never be a one-to-one mapping between vertex coordinates and fragments, so the rasterizer has to determine in some way what fragment/screen-coordinate each specific vertex will end up at.

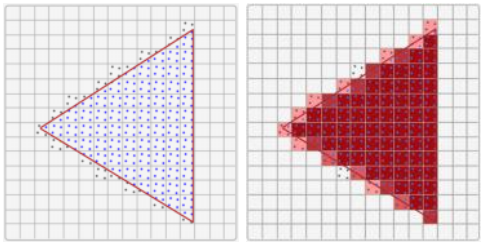
In the image to the right, we see a grid of screen pixels where the center of each pixel contains a **sample point** that is used to determine if a pixel isencompassed by the triangle.

- The red sample points are covered by the triangle, so a fragment will be generated for that pixel.
- Even though some parts of the triangle's edges enter certain screen pixels, the pixel's sample point is not within the triangle, so that pixel will not be influenced by any fragment shader.
- The image to the right of the sampling grid shows what fragments would be rendered for the triangle.
  - Due to the limited amount of screen pixels, some pixels will be rendered along an edge and some won't. The result is that we' re rendering primitives with non-smooth edges, giving rise to the jagged edges, as seen above. As you can see, this is where the issue of aliasing lies.



Instead of using a single sample point for determining coverage of the triangle, multisampling uses multiple.

- Rather than a single sample point at the center of each pixel, we use 4 **subsamples** in a general pattern to determine pixel coverage.
- Looking at the image to the right:
  - The left side of the image shows how we would normally determine pixel coverage.
    - The pixel, in this case, is not covered by the triangle, so it will not run a fragment shader (and will thus remain blank).
  - The right side of the image shows how we would determine pixel coverage using multisampling.
    - The pixel, in this case, is partially covered by the triangle.
    - **NOTE:** We can use any amount of sample point we want to give better coverage precision.



How multisampling handles partial coverage of pixels may be different than you think.

- You might guess that we run the fragment shader for each covered subsample and, later, average the colors of each subsample per pixel. So, in the case above, we'd run the fragment shader twice on the interpolated vertex data at each subsample and store the resulting color in those sample points.
  - This would require much more fragment shader runs and is not how MSAA works.
- MSAA only runs the fragment shader once per pixel (for each primitive), regardless of how many subsamples the triangle covers; the fragment shader runs with the vertex data interpolated to the center of the pixel.
- MSAA then uses a larger depth/stencil buffer to determine subsample coverage. The number of subsamples covered determines how much the pixel color contributes to the framebuffer.
  - Because (in the example above) only 2 of the 4 samples were covered, half of the triangle's color is mixed with the framebuff er color (white), resulting in a light blue color.
  - The result is a higher resolution buffer (with higher resolution depth/stencil) where all the primitive edges now produce a s moother pattern.
    - The hard edges of the triangle are now surrounded by colors slightly lighter than the actual edge color, which causes the edge to appear smooth when viewed from a distance.

Depth and stencil values are stored per subsample and, even though we only run the fragment shader once, color values are stored per subsample as well for the case of multiple triangles overlapping a single pixel.

- For depth testing, the vertex's depth value is interpolated to each subsample before running the depth test.
- For stencil testing, we store the stencil values per subsample.
- This does mean that the size of the buffers are now increased by the amount of subsamples per pixel.

The actual logic behind the rasterizer is a bit more complicated than what is described above, but you should, at this point, understand the concept and logic behind MSAA.

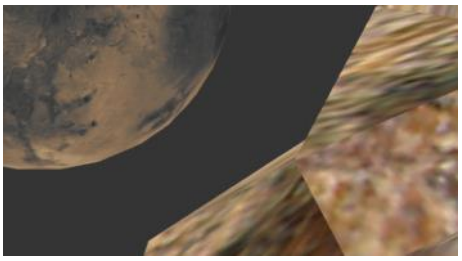
MSAA in OpenGL

In order to use MSAA in OpenGL, we need to use a buffer that can store more than one sample value per pixel; a **multisample buffer**.

- Most windowing systems, including GLFW, are able to provide us a multisample buffer instead of a default buffer.

Specify to GLFW that we'd like to use a multisample buffer with 4 samples instead of a normal buffer. Then, enable multisampling.

- **NOTE:** On most drivers, multisampling is enabled by default, but it's usually a good idea to enable it anyways, just in case.
- GLFW Multisampling Setup



Now, when we create a window (with `glfwCreateWindow`), we create a rendering window with a buffer containing 4 subsamples per screen coordinate.
- Consequentially, the size of the buffer is increased by a factor of 4.
- You should be able to see the effects immediately, since multisampling algorithms are implemented in the rasterizer, which we don't really have the ability to modify.
  - In the image to the right, you may still be able to notice a little jaggedness on the edges, but it has been largely mitigate d by multisampling.

Off-Screen MSAA

Because GLFW takes care of creating the multisampled buffers, enabling MSAA is quite easy. If we want to use our own framebuffers, however, we have to generate the multisampled buffers ourselves.
- There are two ways we can create multisampled buffers to act as attachments for framebuffers (these are quite similar to normal attachments as discussed in the framebuffers chapter):
  1. Texture attachments
  2. Renderbuffer attachments

Multisampled Texture Attachments

Create a texture that supports storage of multiple sample points and attach it to a framebuffer.
- Multisampled Texture Attachment

Multisampled Renderbuffer Attachments

Create a depth and stencil renderbuffer that supports storage of multiple sample points and attach it to a framebuffer.
- Multisampled Renderbuffer Attachment

Render to Multisampled Framebuffer

Rendering to a multisampled framebuffer is as simple binding the multisampled framebuffer and drawing, letting the rasterizer take care of the multisample operations. However, we can't directly use the multisampled buffer for other operations, like sampling it in a shader, like we would normally be able to.
- A **multisampled image** contains much more information than a normal image, so we need to downscale or **resolve** the image.
  - Resolving a multisampled framebuffer is generally done through `glBlitFramebuffer`, which copies a region from one framebuffer to the other while also resolving any multisampled buffers.

`glBlitFramebuffer` transfers a given **source** region, defined by 4 screen-space coordinates, to a given **target** region, also defined by 4 screen-space coordinates.
- The function reads from the GL_READ_FRAMEBUFFER and GL_DRAW_FRAMEBUFFER targets to determine which is the source and which is the target framebuffer.
- You can transfer the multisampled framebuffer output to the screen by **blitting** the framebuffer image to the default framebuffer, like this.

If we want to be able to add post-processing effects, we need to add an intermediate framebuffer to resolve the multisampled framebuffer to.
- This is because we can't sample a multisampled framebuffer's texture in a shader normally, so we essentially map the color values of the multisampled framebuffer's texture to another non-multisampled framebuffer's texture by specifying the intermediate framebuffer as the destination of the blit.

Draw the scene into your multisampled framebuffer. Then, draw the texture of the framebuffer over the whole screen by blitting that framebuffer's color buffer to an intermediate, non-multisampled framebuffer's texture. Finally, test that the various post-processing kernels we used in the framebuffer chapter still work.
- You can use the `screenShader` and `screenQuadVAO` from the framebuffer chapter for this part.
- **NOTE:** Because the screen texture will be a normal, non-multisampled texture again, some post-processing filters like edge-detection will introduce jagged edges again. To accomodate for this, you could blur the texture afterwards or create your own anti-aliasing algorithm.
- MSAA Using Framebuffers

NOTE: Enabling multisampling in your scene can noticeably reduce performance the more samples you use.

Custom Anti-Aliasing Algorithm

Earlier, I said we we can't sample a multisampled framebuffer's texture in a shader *normally*. It is possible to directly pass a multisampled texture image to a fragment shader instead of first resolving it, but it works a little differently than normal.
- GLSL gives us the option to sample the texture image per subsample so we can create our own custom anti-aliasing algorithms.
  - To get a texture value per subsample, you'd have to define the texture uniform sampler like so:
    - `uniform sampler2DMS screenTextureMS;`
  - Using the `texelFetch` function, it is then possible to retrieve the color value per sample, like so:
    - `vec4 colorSample = texelFetch(screenTextureMS, texCoords, 3); // 4th subsample`

We're not going to cover how to create a custom anti-aliasing algorithm here, but you can certainly start working on one of your own with a little more research.

GLFW Multisampling Setup

Monday, May 23, 2022 6:15 PM

```
glfwWindowHint(GLFW_SAMPLES, 4);
...
glEnable(GL_MULTISAMPLE);
```

Multisampled Texture Attachment

Monday, May 23, 2022 6:36 PM

```
unsigned int aaFBO, aaTex;
int numSamples = 4;

// Framebuffer creation
glGenBuffers(1, &aaFBO);
glBindBuffer(GL_FRAMEBUFFER, aaFBO);

// Texture attachment creation
glGenTextures(1, &aaTex);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, aaTex);
// 4 -> # of samples, GL_TRUE -> the image will use identical sample locations and # of subsamples for each texel
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, numSamples, GL_RGB, WIDTH, HEIGHT, GL_TRUE);
glTexParameteri(GL_TEXTURE_2D_MULTISAMPLE, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D_MULTISAMPLE, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);

// Attaching the texture to the framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D_MULTISAMPLE, tex, 0);

// Error checking for if the framebuffer is not complete
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
   std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
}

glBindBuffer(GL_FRAMEBUFFER, 0);
```

This looks very similar to the framebuffer chapter, so if this doesn't make sense, review that chapter.

Multisampled Renderbuffer Attachment

Monday, May 23, 2022 6:52 PM

Below is the code for a multisample renderbuffer attachment...

main()

```
// Depth-stencil Renderbuffer attachment creation
unsigned int aaRBO;
glGenRenderbuffers(1, &aaRBO);
glBindRenderbuffer(GL_RENDERBUFFER, aaRBO);
glRenderbufferStorageMultisample(GL_RENDERBUFFER, numSamples, GL_DEPTH24_STENCIL8, WIDTH, HEIGHT);
glBindRenderbuffer(GL_RENDERBUFFER, 0);

// Attaching the depth-stencil renderbuffer to the framebuffer
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH24_STENCIL8, GL_RENDERBUFFER, aaRBO);
```

... and this is it in combination with the code from the previous subpage.

main()

```
unsigned int aaFBO, aaTex, aaRBO;
int numSamples = 4;

// Framebuffer creation
glGenBuffers(1, &aaFBO);
glBindBuffer(GL_FRAMEBUFFER, aaFBO);

// Texture attachment creation
glGenTextures(1, &aaTex);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, aaTex);
// GL_TRUE -> the image will use identical sample locations and # of subsamples for each texel
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, numSamples, GL_RGB, WIDTH, HEIGHT, GL_TRUE);
glTexParameteri(GL_TEXTURE_2D_MULTISAMPLE, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D_MULTISAMPLE, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);

// Depth-stencil Renderbuffer attachment creation
glGenRenderbuffers(1, &aaRBO);
glBindRenderbuffer(GL_RENDERBUFFER, aaRBO);
glRenderbufferStorageMultisample(GL_RENDERBUFFER, numSamples, GL_DEPTH24_STENCIL8, WIDTH, HEIGHT);
glBindRenderbuffer(GL_RENDERBUFFER, 0);

// Attaching the texture to the framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D_MULTISAMPLE, aaTex, 0);
// Attaching the depth-stencil renderbuffer to the framebuffer
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH24_STENCIL8, GL_RENDERBUFFER, aaRBO);

// Error checking for if the framebuffer is not complete
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
   std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
}

glBindBuffer(GL_FRAMEBUFFER, 0);
```

Multisampled Framebuffer Blitting Example

Monday, May 23, 2022 7:21 PM

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, aaFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBlitFramebuffer(0, 0, WIDTH, HEIGHT, 0, 0, WIDTH, HEIGHT, GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

MSAA Using Framebuffers

Monday, May 23, 2022 9:43 PM

main()

```
unsigned int intermediateFBO, screenTexture;

glGenFramebuffers(1, &intermediateFBO);
glBindFramebuffer(GL_FRAMEBUFFER, intermediateFBO);

glGenTextures(1, &screenTexture);
glBindTexture(GL_TEXTURE_2D, screenTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, WIDTH, HEIGHT, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, screenTexture, 0);

// Error checking for if the framebuffer is not complete
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
   std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
}

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

If the structure of the render loop doesn't make sense, please review the framebuffer chapter.

Render Loop

```
// RENDERING
glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glBindFramebuffer(GL_FRAMEBUFFER, aaFBO);
glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);

// Draw scene
[...]

// Blits the aaFBO color buffer to the intermediateFBO, resolving the color attachment to a standard 2D texture
glBindFramebuffer(GL_READ_FRAMEBUFFER, aaFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, intermediateFBO);
glBlitFramebuffer(0, 0, WIDTH, HEIGHT, 0, 0, WIDTH, HEIGHT, GL_COLOR_BUFFER_BIT, GL_NEAREST);

glBindFramebuffer(GL_FRAMEBUFFER, 0);

// Clears the framebuffer color
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

glDisable(GL_DEPTH_TEST); // Depth testing is disabled so the texture below is drawn over everything

// Draws the resolved color attachment of the intermediateFBO framebuffer to a quad spanning the screen
screenShader.use();
glBindVertexArray(screenQuadVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, screenTexture); // Use the now resolved color attachment as the quad's texture
glUniform1i(glGetUniformLocation(screenShader.id, "screenTexture"), 0);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

The `screenShader` and `screenQuadVAO` are the exact same from the framebuffer chapter. However, I'll put the code for them below as well:

screen.vert

```
#version 330 core

layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 texCoords;

void main() {
   gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
   texCoords = aTexCoords;
}
```

If you want to view any of the post processing effects using the kernels, just uncomment one and comment out the rest.

screen.frag

```
#version 330 core

out vec4 FragColor;

in vec2 texCoords;

uniform sampler2D screenTexture;

const float offset = 1.0 / 300.0;

void main() {
    // Offset values that are used in determining what adjacent fragments to sample
    vec2 offsets[9] = vec2[](
        vec2(-offset,  offset), // top-left
        vec2( 0.0f,    offset), // top-center
        vec2( offset,  offset), // top-right
        vec2(-offset,  0.0f),   // center-left
        vec2( 0.0f,    0.0f),   // center-center
        vec2( offset,  0.0f),   // center-right
        vec2(-offset, -offset), // bottom-left
        vec2( 0.0f,   -offset), // bottom-center
        vec2( offset, -offset)  // bottom-right
    );

    // Default kernel (has no effect)
    float kernel[9] = float[](
        0, 0, 0,
        0, 1, 0,
        0, 0, 0
    );

    // Sharpen kernel
    /*float kernel[9] = float[](
        -1.0,  -1.0, -1.0,
        -1.0,   9.0, -1.0,
        -1.0,  -1.0, -1.0
    );*/

    // Blur kernel
    /*float kernel[9] = float[](
        1.0 / 16.0,  2.0 / 16.0, 1.0 / 16.0,
        2.0 / 16.0,  4.0 / 16.0, 2.0 / 16.0,
        1.0 / 16.0,  2.0 / 16.0, 1.0 / 16.0
    );*/

    // Edge-detection kernel
    /*float kernel[9] = float[](
        1.0,  1.0, 1.0,
        1.0, -8.0, 1.0,
        1.0,  1.0, 1.0
```

```
    );*/

    // Stores the adjacent fragment colors for kernelling
    vec3 sampleTex[9];
    for (int i = 0; i < 9; ++i) {
        sampleTex[i] = vec3(texture(screenTexture, texCoords.st + offsets[i]));
    }

    // Applies the kernel to the adjacent fragments' colors to determine this fragment's color
    vec3 color = vec3(0.0);
    for (int i = 0; i < 9; ++i) {
        color += sampleTex[i] * kernel[i];
    }

    FragColor = vec4(color, 1.0);
}
```

main.cpp

```
float screenQuadVertices[] = {
   // Positions      // Tex Coords
   -1.0f, -1.0f,     0.0f, 0.0f,       // bottom left
    1.0f,  1.0f,     1.0f, 1.0f,       // top right
   -1.0f,  1.0f,     0.0f, 1.0f,       // top left

   -1.0f, -1.0f,     0.0f, 0.0f,       // bottom left
    1.0f, -1.0f,     1.0f, 0.0f,       // bottom right
    1.0f,  1.0f,     1.0f, 1.0f        // top right
};
```

main()

```
Shader screenShader{ "Shaders/screen.vert", "Shaders/screen.frag" };

...

unsigned int screenQuadVAO, screenQuadVBO;
glGenVertexArrays(1, &screenQuadVAO);
glGenBuffers(1, &screenQuadVBO);

glBindVertexArray(screenQuadVAO);
glBindBuffer(GL_ARRAY_BUFFER, screenQuadVBO);

glBufferData(GL_ARRAY_BUFFER, sizeof(screenQuadVertices), &screenQuadVertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)0);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(sizeof(float) * 2));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```