Aliasing

“Aliasing” is a signal-processing term for “under-sampled compared with the frequencies in the signal.”

What the signal really is:
what we want

What we think the signal is:
too often, what we get

Sampling Interval

Sampled Points

The Display We Want

Too often, the Display We Get
Nyquist Criterion

“The Nyquist [sampling] rate is twice the maximum component frequency of the function [i.e., signal] being sampled.” -- Wikipedia

Multisampling

Multisampling is a computer graphics technique to improve the quality of your output image by looking inside every pixel to see what the rendering is doing there. There are two approaches:

1. **Supersampling**: Pick some number of unique sub-pixels within a pixel, render the image at each of these individual sub-pixels (including depth and stencil tests), then average them together. This results in lots of renders.

2. **Multisampling**: Perform a single color render for the one entire pixel. Then, pick some number of unique sub-pixels within that pixel and perform depth and stencil tests there. Assign the single color to all the sub-pixels that made it through the depth and stencil tests.

Note: per-sample depth and stencil tests are performed first to decide which color renders actually should be done.
Vulkan Distribution of Sampling Points within a Pixel

<table>
<thead>
<tr>
<th>VK_SAMPLE_COUNT_1_BIT</th>
<th>VK_SAMPLE_COUNT_2_BIT</th>
<th>VK_SAMPLE_COUNT_4_BIT</th>
<th>VK_SAMPLE_COUNT_8_BIT</th>
<th>VK_SAMPLE_COUNT_16_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16,64)</td>
<td>(32,128)</td>
<td>(64,256)</td>
<td>(128,512)</td>
<td>(256,1024)</td>
</tr>
<tr>
<td>(32,128)</td>
<td>(64,256)</td>
<td>(128,512)</td>
<td>(256,1024)</td>
<td>(512,2048)</td>
</tr>
<tr>
<td>(64,256)</td>
<td>(128,512)</td>
<td>(256,1024)</td>
<td>(512,2048)</td>
<td>(1024,4096)</td>
</tr>
<tr>
<td>(128,512)</td>
<td>(256,1024)</td>
<td>(512,2048)</td>
<td>(1024,4096)</td>
<td>(2048,8192)</td>
</tr>
<tr>
<td>(256,1024)</td>
<td>(512,2048)</td>
<td>(1024,4096)</td>
<td>(2048,8192)</td>
<td>(4096,16384)</td>
</tr>
<tr>
<td>(512,2048)</td>
<td>(1024,4096)</td>
<td>(2048,8192)</td>
<td>(4096,16384)</td>
<td>(8192,32768)</td>
</tr>
<tr>
<td>(1024,4096)</td>
<td>(2048,8192)</td>
<td>(4096,16384)</td>
<td>(8192,32768)</td>
<td>(16384,65536)</td>
</tr>
</tbody>
</table>

Consider Two Triangles Whose Edges Pass Through the Same Pixel

**Supersampling**

Final Pixel Color = \[\sum \text{Color sample from subpixel} \]

# Fragment Shader calls = 8

**Multisampling**

Final Pixel Color = 3 \cdot \text{One color sample from A} + 5 \cdot \text{One color sample from B}

# Fragment Shader calls = 2
Setting up the Image

VkPipelineMultisampleStateCreateInfo

vpmsci;
vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmsci.pNext = nullptr;
vpmsci.flags = 0;
vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;
vpmsci.sampleShadingEnable = VK_TRUE;
vpmsci.minSampleShading = 0.5f;
vpmsci.pSampleMask = (VkSampleMask *)nullptr;
vpmsci.alphaToCoverageEnable = VK_FALSE;
vpmsci.alphaToOneEnable = VK_FALSE;

VkGraphicsPipelineCreateInfo

vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.pMultisampleState = &vpmsci;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline);

Setting up the Image

VkAttachmentDescription

vad[2];
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[0].samples = VK_SAMPLE_COUNT_8_BIT;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
vad[0].flags = 0;

vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline);

Setting up the Image

VkSubpassDescription

vsd;
vsd.flags = 0;
vsd.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd.inputAttachmentCount = 0;
vsd.pInputAttachments = (VkAttachmentReference *)nullptr;
vsd.colorAttachmentCount = 1;
vsd.pColorAttachments = &colorReference;
vsd.colorAttachmentCount = 1;
vsd.pColorAttachments = &colorReference;
vsd.pDepthStencilAttachment = &depthReference;
vsd.preserveAttachmentCount = 0;
vsd.pPreserveAttachments = (uint32_t *)nullptr;
result = vkCreateRenderPass( LogicalDevice, IN &vpri, PALLOCATOR, OUT &RenderPass );

At least this fraction of samples will get their own fragment shader calls (as long as they pass the depth and stencil tests).
0. produces simple multisampling
0.1. produces partial supersampling
1. produces complete supersampling
Resolving the Image:
Converting the multisampled image to a VK_SAMPLE_COUNT_1_BIT image

```c
VOffset3D vo3;
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;

VkExtent3D ve3;
ve3.width = Width;
ve3.height = Height;
ve3.depth = 1;

VkImageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visl.mipLevel = 0;
visl.baseArrayLayer = 0;
visl.layerCount = 1;

VkImageResolve vir;
vir.srcSubresource = visl;
vir.srcOffset = vo3;
vir.dstSubresource = visl;
vir.dstOffset = vo3;
vir.extent = ve3;

vkCmdResolveImage(cmdBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, 1, &vir);
```