Antialiasing and Multisampling

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

Aliasing

The Display We Want

Too often, the Display We Get
“Aliasing” is a signal-processing term for “under-sampled compared with the frequencies in the signal”.

What the signal really is: what we want

Sampling Interval

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

Sampled Points
The Nyquist Criterion

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

MultiSampling

Oversampling 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 to this:

1. Supersampling: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. Render the image at each of these sub-pixels.

2. Multisampling: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. If any of them pass, then perform a single color render for the one pixel and assign that single color to all the sub-pixels that passed the depth and stencil tests.

The final step will be to average those sub-pixels' colors to produce one final color for this whole pixel. This is called resolving the pixel.
### Vulkan Specification Distribution of Sampling Points within a Pixel

#### Table: Sampling Point Distribution

<table>
<thead>
<tr>
<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>(0.375, 0.125)</td>
<td>(0.5625, 0.3125)</td>
<td>(0.30, 0.25)</td>
<td>(0.625, 0.875)</td>
</tr>
<tr>
<td>(0.4375, 0.6875)</td>
<td>(0.3125, 0.625)</td>
<td>(0.1875, 0.375)</td>
<td>(0.875, 0.375)</td>
</tr>
<tr>
<td>(0.3125, 0.1875)</td>
<td>(0.625, 0.8125)</td>
<td>(0.8125, 0.6875)</td>
<td>(0.125, 0.625)</td>
</tr>
<tr>
<td>(0.1875, 0.8125)</td>
<td>(0.5, 0.0625)</td>
<td>(0.0625, 0.4375)</td>
<td>(0.125, 0.75)</td>
</tr>
<tr>
<td>(0.375, 0.9375)</td>
<td>(0.125, 0.75)</td>
<td>(0.6875, 0.9375)</td>
<td>(0.0, 0)</td>
</tr>
<tr>
<td>(0.625, 0.875)</td>
<td>(0.625, 0.0625)</td>
<td>(0.9375, 0.25)</td>
<td>(0.875, 0.9375)</td>
</tr>
<tr>
<td>(0.625, 0.0625)</td>
<td></td>
<td>(0.0625, 0.0)</td>
<td></td>
</tr>
</tbody>
</table>
Consider Two Triangles Who Pass Through the Same Pixel

Let's assume (for now) that the two triangles don't overlap – that is, they look this way because they butt up against each other.

Supersampling

\[
\text{Final Pixel Color} = \frac{\sum_{i=1}^{8} \text{Color sample from subpixel}_i}{8}
\]

# Fragment Shader calls = 8
Consider Two Triangles Who Pass Through the Same Pixel

Let's assume (for now) that the two triangles don't overlap – that is, they look this way because they butt up against each other.

<table>
<thead>
<tr>
<th></th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>shader calls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Red fragment</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>shader calls</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Final Pixel Color = \frac{3 \cdot \text{One color sample from A} + 5 \cdot \text{One color sample from B}}{8}
Q: What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?

A: The ideas are all still the same, but the blue one had to deal with 8 sub-pixels (instead of 5 like before). But, the red triangle came along and obsoleted 3 of those blue sub-pixels. Note that the "resolved" image will still turn out the same as before.

Consider Two Triangles Who Pass Through the Same Pixel

What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?

<table>
<thead>
<tr>
<th></th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment shader calls</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Red fragment shader calls</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Setting up the Image

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

```
VkPipelineMultisampleStateCreateInfo vpmsci;

vpmsci.minSampleShading = 0.5f;

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
```
### Setting up the Image

```c
// VkAttachmentDescription
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_ATTACHMENTSTORE Op_STORE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE Op DONT_CARE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
vad[0].flags = 0;

vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[1].samples = VK_SAMPLE_COUNT_8_BIT;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
vad[1].flags = 0;

// VkAttachmentReference
vkAttachmentReference colorReference;
colorReference.attachment = 0;
colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

vkAttachmentReference depthReference;
depthReference.attachment = 1;
depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

### Setting up the Image

```c
// VkSubpassDescription
vsd:
vsd.flags = 0;
vsd.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd.inputAttachmentCount = 0;
vsd.pInputAttachments = (VkAttachmentReference *)nullptr;
vsd.colorAttachmentCount = 1;  // color
vsd.pColorAttachments = &colorReference;
vsd.pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd.pDepthStencilAttachment = &depthReference;
vsd.preserveAttachmentCount = 0;
vsd.pPreserveAttachments = (uint32_t *)nullptr;
```

```c
// VkRenderPassCreateInfo
vkRenderPassCreateInfo vrpci;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.flags = 0;
vrpci.pAttachments = vad;
vrpci.pSubpasses = 1;
vrpci.pSubpasses = 1;  // IN & vad;
vrpci.dependencyCount = 0;
vrpci.pDependencies = (VkSubpassDependency *)nullptr;
```

```c
result = vkCreateRenderPass( LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass );
```

Oregon State
University
Computer Graphics

mjb – January 30, 2020
Resolving the Image:
Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

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

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

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

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

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

For the *ImageLayout, use VK_IMAGE_LAYOUT_GENERAL