The current eye position
- the current light source positions and colors
- texture coordinates (s,t)
- material color (r,g,b)
- normal (nx,ny,nz)
- position (x,y,z)

Each fragment’s:
- the current eye position
- lighting model
- pixels that can be seen

When we write these out, the final framebuffers will contain just information for those pixels. This known as the G-buffer Algorithm.

What is an Example of Wanting to do This?

There is a process in computer graphics called **Deferred Rendering**. The idea is that a game-quality fragment shader takes a long time (relatively) to execute, but, with all the 3D scene detail, a lot of the rendered fragments are going to get z-buffered away anyhow. So, why did we invoke the fragment shaders so many times when we didn’t need to?

Here’s the trick:

Let’s create a grossly simple fragment shader that writes out (into multiple framebuffers)

Here comes a quick reminder of how we did that. Afterwards, we will extend it.

Back in Our Single-pass Days, I

There is a process in computer graphics called **Deferred Rendering**. The idea is that a game-quality fragment shader takes a long time (relatively) to execute, but, with all the 3D scene detail, a lot of the rendered fragments are going to get z-buffered away anyhow. So, why did we invoke the fragment shaders so many times when we didn’t need to?

Here’s the trick:

Let’s create a grossly simple fragment shader that writes out (into multiple framebuffers)

As well as:
- the current light source positions and colors
- the current eye position

Here comes a quick reminder of how we did that. Afterwards, we will extend it.

Back in Our Single-pass Days, II

Let’s create a grossly simple fragment shader that writes out (into multiple framebuffers)
In this case, we will look at following up a 3D rendering with Gbuffer operations.

```cpp
VkAttachmentReference gbufferInput;
VkAttachmentReference depthOutput;
VkAttachmentReference gbufferOutput;
VkAttachmentReference lightingOutput;
```

Subpass #0 Subpass #1

```cpp
gBufferInput.attachment = 0; // depth
depthOutput.attachment = 0; // depth
gBufferOutput.attachment = 1; // gbuffer
lightingOutput.attachment = 2; // color rendering
lightingInput[0].layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
lightingInput[0].attachment = 0; // depth
lightingInput[1].attachment = 1; // gbuffer
```

```cpp
vsdp[0].srcSubpass = 0; // depth rendering
vsdp[1].dstSubpass = 2; //
vsdp[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
vsdp[0].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vsdp[0].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
vsdp[0].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
vsdp[1].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
vsdp[1].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
vsdp[1].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
```

```cpp
vkCreateRenderPass
vrpci.pNext = nullptr;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pDependencies = nullptr;
vrpci.dependencyCount = 2;
vrpci.pSubpasses = vsdp;
vrpci.subpassCount = 3;
vrpci.pAttachments = vad;
```

```cpp
vsd[0].preserveAttachmentCount = 0;
vsd[0].pDepthStencilAttachment = (VkAttachmentReference *) nullptr;
vsd[0].pResolveAttachments = (VkAttachmentReference *) nullptr;
vsd[0].pColorAttachments = [3];
vsd[0].colorAttachmentCount = 3;
vsd[0].inputAttachmentCount = 2;
vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

```cpp
vsd[1].preserveAttachmentCount = 0;
vsd[1].pDepthStencilAttachment = (VkAttachmentReference *) nullptr;
vsd[1].pResolveAttachments = (VkAttachmentReference *) nullptr;
vsd[1].pColorAttachments = (VkAttachmentReference *) nullptr;
vsd[1].pInputAttachments = [3];
vsd[1].inputAttachmentCount = 3;
vsd[1].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

```cpp
vsd[2].preserveAttachmentCount = 0;
vsd[2].pDepthStencilAttachment = [3];
vsd[2].pResolveAttachments = [3];
vsd[2].pColorAttachments = [3];
vsd[2].pInputAttachments = [3];
```

```cpp
vad[0].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[0].flags = 0;
```

```cpp
vad[1].format = VK_FORMAT_R8G8B8A8_SRGB;
```

```cpp
vad[2].format = VK_FORMAT_R16G16B16A16_SFLOAT;
```

```cpp
vad[0].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
```

```cpp
vad[1].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
```

```cpp
vad[2].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
vad[2].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
```

```cpp
vad[2].samples = VK_SAMPLE_COUNT_1_BIT;
```

```cpp
Multipass, I
```

```cpp
Multipass, II
```

```cpp
Multipass, III
```

```cpp
Multipass, IV
```

```cpp
Multipass, V
```
vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);

// subpass #0 is automatically started here
vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);

vkCmdBindDescriptorSets(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *) nullptr);

vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);

vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

// . . .

vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);

// subpass #1 is started here

// . . .

vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);

// subpass #2 is started here

// . . .

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);