

**GraphicsEngine**

Generated by Doxygen 1.9.2



---

<b>1 Namespace Index</b>	<b>1</b>
1.1 Namespace List . . . . .	1
<b>2 Data Structure Index</b>	<b>3</b>
2.1 Data Structures . . . . .	3
<b>3 File Index</b>	<b>5</b>
3.1 File List . . . . .	5
<b>4 Namespace Documentation</b>	<b>9</b>
4.1 debug Namespace Reference . . . . .	9
4.1.1 Function Documentation . . . . .	9
4.1.1.1 debugUtilsMessengerCallback() . . . . .	10
4.1.1.2 freeDebugCallback() . . . . .	11
4.1.1.3 messageCallback() . . . . .	11
4.1.1.4 setupDebugging() . . . . .	11
4.1.2 Variable Documentation . . . . .	12
4.1.2.1 debugUtilsMessenger . . . . .	12
4.1.2.2 validationLayerCount . . . . .	13
4.1.2.3 validationLayerNames . . . . .	13
4.1.2.4 vkCreateDebugUtilsMessengerEXT . . . . .	13
4.1.2.5 vkDestroyDebugUtilsMessengerEXT . . . . .	13
4.2 sceneConfig Namespace Reference . . . . .	13
4.2.1 Function Documentation . . . . .	13
4.2.1.1 getModelFile() . . . . .	14
4.2.1.2 getModelMatrix() . . . . .	14
4.3 std Namespace Reference . . . . .	15
4.4 vertex Namespace Reference . . . . .	15
4.4.1 Function Documentation . . . . .	15
4.4.1.1 getVertexInputAttributeDesc() . . . . .	16
<b>5 Data Structure Documentation</b>	<b>17</b>
5.1 Allocator Class Reference . . . . .	17
5.1.1 Detailed Description . . . . .	18
5.1.2 Constructor & Destructor Documentation . . . . .	18
5.1.2.1 Allocator() [1/2] . . . . .	18
5.1.2.2 Allocator() [2/2] . . . . .	18
5.1.2.3 ~Allocator() . . . . .	18
5.1.3 Member Function Documentation . . . . .	19
5.1.3.1 cleanUp() . . . . .	19
5.1.4 Field Documentation . . . . .	19
5.1.4.1 vmaAllocator . . . . .	19
5.2 App Class Reference . . . . .	20
5.2.1 Detailed Description . . . . .	20

---

5.2.2 Constructor & Destructor Documentation . . . . .	20
5.2.2.1 App() . . . . .	20
5.2.2.2 ~App() . . . . .	20
5.2.3 Member Function Documentation . . . . .	21
5.2.3.1 run() . . . . .	21
5.3 ASManager Class Reference . . . . .	22
5.3.1 Detailed Description . . . . .	23
5.3.2 Constructor & Destructor Documentation . . . . .	23
5.3.2.1 ASManager() . . . . .	23
5.3.2.2 ~ASManager() . . . . .	23
5.3.3 Member Function Documentation . . . . .	23
5.3.3.1 cleanUp() . . . . .	24
5.3.3.2 createAccelerationStructureInfosBLAS() . . . . .	25
5.3.3.3 createASForScene() . . . . .	26
5.3.3.4 createBLAS() . . . . .	27
5.3.3.5 createSingleBlas() . . . . .	29
5.3.3.6 createTLAS() . . . . .	31
5.3.3.7 getTLAS() . . . . .	34
5.3.3.8 objectToVkGeometryKHR() . . . . .	35
5.3.4 Field Documentation . . . . .	36
5.3.4.1 blas . . . . .	36
5.3.4.2 commandBufferManager . . . . .	36
5.3.4.3 tlas . . . . .	37
5.3.4.4 vulkanBufferManager . . . . .	37
5.3.4.5 vulkanDevice . . . . .	37
5.4 BlasInput Struct Reference . . . . .	37
5.4.1 Detailed Description . . . . .	38
5.4.2 Field Documentation . . . . .	38
5.4.2.1 as_build_offset_info . . . . .	38
5.4.2.2 as_geometry . . . . .	38
5.5 BottomLevelAccelerationStructure Struct Reference . . . . .	39
5.5.1 Detailed Description . . . . .	39
5.5.2 Field Documentation . . . . .	40
5.5.2.1 vulkanAS . . . . .	40
5.5.2.2 vulkanBuffer . . . . .	40
5.6 BuildAccelerationStructure Struct Reference . . . . .	40
5.6.1 Detailed Description . . . . .	41
5.6.2 Field Documentation . . . . .	41
5.6.2.1 build_info . . . . .	42
5.6.2.2 range_info . . . . .	42
5.6.2.3 single_bla . . . . .	42
5.6.2.4 size_info . . . . .	42

---

5.7 Camera Class Reference . . . . .	43
5.7.1 Detailed Description . . . . .	44
5.7.2 Constructor & Destructor Documentation . . . . .	44
5.7.2.1 Camera() . . . . .	45
5.7.2.2 ~Camera() . . . . .	45
5.7.3 Member Function Documentation . . . . .	45
5.7.3.1 calculate_viewmatrix() . . . . .	45
5.7.3.2 get_camera_direction() . . . . .	46
5.7.3.3 get_camera_position() . . . . .	46
5.7.3.4 get_far_plane() . . . . .	47
5.7.3.5 get_fov() . . . . .	47
5.7.3.6 get_near_plane() . . . . .	47
5.7.3.7 get_right_axis() . . . . .	48
5.7.3.8 get_up_axis() . . . . .	48
5.7.3.9 get_yaw() . . . . .	48
5.7.3.10 key_control() . . . . .	48
5.7.3.11 mouse_control() . . . . .	49
5.7.3.12 set_camera_position() . . . . .	49
5.7.3.13 set_far_plane() . . . . .	50
5.7.3.14 set_fov() . . . . .	50
5.7.3.15 set_near_plane() . . . . .	50
5.7.3.16 update() . . . . .	50
5.7.4 Field Documentation . . . . .	51
5.7.4.1 far_plane . . . . .	51
5.7.4.2 fov . . . . .	51
5.7.4.3 front . . . . .	51
5.7.4.4 movement_speed . . . . .	52
5.7.4.5 near_plane . . . . .	52
5.7.4.6 pitch . . . . .	52
5.7.4.7 position . . . . .	52
5.7.4.8 right . . . . .	52
5.7.4.9 turn_speed . . . . .	53
5.7.4.10 up . . . . .	53
5.7.4.11 world_up . . . . .	53
5.7.4.12 yaw . . . . .	53
5.8 CommandBufferManager Class Reference . . . . .	54
5.8.1 Detailed Description . . . . .	54
5.8.2 Constructor & Destructor Documentation . . . . .	54
5.8.2.1 CommandBufferManager() . . . . .	54
5.8.2.2 ~CommandBufferManager() . . . . .	55
5.8.3 Member Function Documentation . . . . .	55
5.8.3.1 beginCommandBuffer() . . . . .	55

5.8.3.2 endAndSubmitCommandBuffer()	56
5.9 File Class Reference	57
5.9.1 Detailed Description	57
5.9.2 Constructor & Destructor Documentation	57
5.9.2.1 File()	58
5.9.2.2 ~File()	58
5.9.3 Member Function Documentation	58
5.9.3.1 getBaseDir()	58
5.9.3.2 read()	59
5.9.3.3 readCharSequence()	59
5.9.4 Field Documentation	60
5.9.4.1 file_location	60
5.10 GlobalUBO Struct Reference	60
5.10.1 Detailed Description	61
5.10.2 Field Documentation	61
5.10.2.1 projection	61
5.10.2.2 view	62
5.11 GUI Class Reference	62
5.11.1 Detailed Description	63
5.11.2 Constructor & Destructor Documentation	63
5.11.2.1 GUI()	63
5.11.2.2 ~GUI()	63
5.11.3 Member Function Documentation	63
5.11.3.1 cleanUp()	64
5.11.3.2 create_fonts_and_upload()	64
5.11.3.3 create_gui_context()	65
5.11.3.4 getGuiRendererSharedVars()	67
5.11.3.5 getGuiSceneSharedVars()	68
5.11.3.6 initializeVulkanContext()	68
5.11.3.7 render()	69
5.11.4 Field Documentation	71
5.11.4.1 commandBufferManager	71
5.11.4.2 device	71
5.11.4.3 gui_descriptor_pool	71
5.11.4.4 guiRendererSharedVars	71
5.11.4.5 guiSceneSharedVars	72
5.11.4.6 window	72
5.12 GUIRendererSharedVars Struct Reference	72
5.12.1 Detailed Description	73
5.12.2 Field Documentation	73
5.12.2.1 pathTracing	73
5.12.2.2 raytracing	73

---

5.12.2.3 <code>shader_hot_reload_triggered</code>	73
5.13 <code>GUISceneSharedVars</code> Struct Reference	74
5.13.1 Detailed Description	74
5.13.2 Field Documentation	74
5.13.2.1 <code>direcional_light_radiance</code>	74
5.13.2.2 <code>directional_light_color</code>	75
5.13.2.3 <code>directional_light_direction</code>	75
5.14 <code>std::hash&lt; Vertex &gt;</code> Struct Reference	75
5.14.1 Detailed Description	75
5.14.2 Member Function Documentation	76
5.14.2.1 <code>operator()</code>	76
5.15 Mesh Class Reference	76
5.15.1 Detailed Description	78
5.15.2 Constructor & Destructor Documentation	78
5.15.2.1 <code>Mesh() [1/2]</code>	79
5.15.2.2 <code>Mesh() [2/2]</code>	80
5.15.2.3 <code>~Mesh()</code>	80
5.15.3 Member Function Documentation	80
5.15.3.1 <code>cleanUp()</code>	81
5.15.3.2 <code>createIndexBuffer()</code>	81
5.15.3.3 <code>createMaterialBuffer()</code>	82
5.15.3.4 <code>createMaterialIDBuffer()</code>	83
5.15.3.5 <code>createVertexBuffer()</code>	84
5.15.3.6 <code>getIndexBuffer()</code>	85
5.15.3.7 <code>getIndexCount()</code>	86
5.15.3.8 <code>getMaterialIDBuffer()</code>	86
5.15.3.9 <code>getModel()</code>	87
5.15.3.10 <code>getObjectDescription()</code>	87
5.15.3.11 <code>getVertexBuffer()</code>	87
5.15.3.12 <code>getVertexCount()</code>	88
5.15.3.13 <code>setModel()</code>	88
5.15.4 Field Documentation	88
5.15.4.1 <code>device</code>	88
5.15.4.2 <code>index_count</code>	89
5.15.4.3 <code>indexBuffer</code>	89
5.15.4.4 <code>materialIdsBuffer</code>	89
5.15.4.5 <code>materialsBuffer</code>	89
5.15.4.6 <code>model</code>	89
5.15.4.7 <code>object_description</code>	90
5.15.4.8 <code>objectDescriptionBuffer</code>	90
5.15.4.9 <code>vertex_count</code>	90
5.15.4.10 <code>vertexBuffer</code>	90

5.15.4.11 vulkanBufferManager	90
5.16 Model Class Reference	91
5.16.1 Detailed Description	92
5.16.2 Constructor & Destructor Documentation	92
5.16.2.1 Model() [1/2]	92
5.16.2.2 Model() [2/2]	93
5.16.2.3 ~Model()	93
5.16.3 Member Function Documentation	93
5.16.3.1 add_new_mesh()	93
5.16.3.2 addSampler()	94
5.16.3.3 addTexture()	95
5.16.3.4 cleanUp()	95
5.16.3.5 getCustomInstanceIndex()	96
5.16.3.6 getMesh()	96
5.16.3.7 getMeshCount()	96
5.16.3.8 getModel()	96
5.16.3.9 getObjectDescription()	97
5.16.3.10 getPrimitiveCount()	97
5.16.3.11 getTextureCount()	98
5.16.3.12 getTextureList()	98
5.16.3.13 getTextures()	98
5.16.3.14 getTextureSamplers()	98
5.16.3.15 set_model()	98
5.16.4 Field Documentation	99
5.16.4.1 device	99
5.16.4.2 mesh	99
5.16.4.3 mesh_model_index	99
5.16.4.4 model	99
5.16.4.5 modelTextures	99
5.16.4.6 modelTextureSamplers	100
5.16.4.7 texture_list	100
5.17 ObjectDescription Struct Reference	100
5.17.1 Detailed Description	101
5.17.2 Field Documentation	101
5.17.2.1 index_address	101
5.17.2.2 material_address	101
5.17.2.3 material_index_address	101
5.17.2.4 vertex_address	102
5.18 ObjLoader Class Reference	102
5.18.1 Detailed Description	103
5.18.2 Constructor & Destructor Documentation	103
5.18.2.1 ObjLoader()	103

5.18.3 Member Function Documentation . . . . .	103
5.18.3.1 loadModel() . . . . .	104
5.18.3.2 loadTexturesAndMaterials() . . . . .	105
5.18.3.3 loadVertices() . . . . .	106
5.18.4 Field Documentation . . . . .	108
5.18.4.1 command_pool . . . . .	108
5.18.4.2 device . . . . .	108
5.18.4.3 indices . . . . .	108
5.18.4.4 materialIndex . . . . .	109
5.18.4.5 materials . . . . .	109
5.18.4.6 textures . . . . .	109
5.18.4.7 transfer_queue . . . . .	109
5.18.4.8 vertices . . . . .	109
5.19 ObjMaterial Struct Reference . . . . .	110
5.19.1 Detailed Description . . . . .	110
5.19.2 Field Documentation . . . . .	110
5.19.2.1 ambient . . . . .	111
5.19.2.2 diffuse . . . . .	111
5.19.2.3 dissolve . . . . .	111
5.19.2.4 emission . . . . .	111
5.19.2.5 illum . . . . .	111
5.19.2.6 ior . . . . .	111
5.19.2.7 shininess . . . . .	112
5.19.2.8 specular . . . . .	112
5.19.2.9 textureID . . . . .	112
5.19.2.10 transmittance . . . . .	112
5.20 PathTracing Class Reference . . . . .	112
5.20.1 Detailed Description . . . . .	113
5.20.2 Constructor & Destructor Documentation . . . . .	113
5.20.2.1 PathTracing() . . . . .	114
5.20.2.2 ~PathTracing() . . . . .	114
5.20.3 Member Function Documentation . . . . .	114
5.20.3.1 cleanUp() . . . . .	114
5.20.3.2 createPipeline() . . . . .	115
5.20.3.3 createQueryPool() . . . . .	116
5.20.3.4 init() . . . . .	117
5.20.3.5 recordCommands() . . . . .	118
5.20.3.6 shaderHotReload() . . . . .	120
5.20.4 Field Documentation . . . . .	121
5.20.4.1 . . . . .	121
5.20.4.2 device . . . . .	121
5.20.4.3 maxComputeWorkGroupCount . . . . .	122

5.20.4.4 maxComputeWorkGroupInvocations . . . . .	122
5.20.4.5 maxComputeWorkGroupSize . . . . .	122
5.20.4.6 pathTracingTiming . . . . .	122
5.20.4.7 pc_range . . . . .	122
5.20.4.8 pipeline . . . . .	123
5.20.4.9 pipeline_layout . . . . .	123
5.20.4.10 push_constant . . . . .	123
5.20.4.11 query_count . . . . .	123
5.20.4.12 queryPool . . . . .	123
5.20.4.13 queryResults . . . . .	124
5.20.4.14 specializationData . . . . .	124
5.20.4.15 timeStampPeriod . . . . .	124
5.21 PostStage Class Reference . . . . .	124
5.21.1 Detailed Description . . . . .	126
5.21.2 Constructor & Destructor Documentation . . . . .	126
5.21.2.1 PostStage() . . . . .	126
5.21.2.2 ~PostStage() . . . . .	126
5.21.3 Member Function Documentation . . . . .	126
5.21.3.1 cleanUp() . . . . .	127
5.21.3.2 createDepthbufferImage() . . . . .	127
5.21.3.3 createFramebuffer() . . . . .	128
5.21.3.4 createGraphicsPipeline() . . . . .	130
5.21.3.5 createOffscreenTextureSampler() . . . . .	133
5.21.3.6 createPushConstantRange() . . . . .	134
5.21.3.7 createRenderpass() . . . . .	134
5.21.3.8 getOffscreenSampler() . . . . .	136
5.21.3.9 getRenderPass() . . . . .	137
5.21.3.10 init() . . . . .	137
5.21.3.11 recordCommands() . . . . .	138
5.21.3.12 shaderHotReload() . . . . .	140
5.21.4 Field Documentation . . . . .	141
5.21.4.1 depth_format . . . . .	141
5.21.4.2 depthBufferImage . . . . .	141
5.21.4.3 device . . . . .	141
5.21.4.4 framebuffers . . . . .	141
5.21.4.5 graphics_pipeline . . . . .	142
5.21.4.6 offscreenTextureSampler . . . . .	142
5.21.4.7 pipeline_layout . . . . .	142
5.21.4.8 push_constant_range . . . . .	142
5.21.4.9 render_pass . . . . .	142
5.21.4.10 vulkanSwapChain . . . . .	143
5.22 PushConstantPathTracing Struct Reference . . . . .	143

---

5.22.1 Detailed Description . . . . .	143
5.22.2 Field Documentation . . . . .	144
5.22.2.1 clearColor . . . . .	144
5.22.2.2 height . . . . .	144
5.22.2.3 width . . . . .	144
5.23 PushConstantPost Struct Reference . . . . .	145
5.23.1 Detailed Description . . . . .	145
5.23.2 Field Documentation . . . . .	145
5.23.2.1 aspect_ratio . . . . .	145
5.24 PushConstantRasterizer Struct Reference . . . . .	146
5.24.1 Detailed Description . . . . .	146
5.24.2 Field Documentation . . . . .	146
5.24.2.1 model . . . . .	146
5.25 PushConstantRaytracing Struct Reference . . . . .	147
5.25.1 Detailed Description . . . . .	147
5.25.2 Field Documentation . . . . .	147
5.25.2.1 clear_color . . . . .	147
5.26 QueueFamilyIndices Struct Reference . . . . .	148
5.26.1 Detailed Description . . . . .	148
5.26.2 Member Function Documentation . . . . .	148
5.26.2.1 is_valid() . . . . .	149
5.26.3 Field Documentation . . . . .	149
5.26.3.1 compute_family . . . . .	149
5.26.3.2 graphics_family . . . . .	149
5.26.3.3 presentation_family . . . . .	150
5.27 Rasterizer Class Reference . . . . .	150
5.27.1 Detailed Description . . . . .	151
5.27.2 Constructor & Destructor Documentation . . . . .	151
5.27.2.1 Rasterizer() . . . . .	151
5.27.2.2 ~Rasterizer() . . . . .	152
5.27.3 Member Function Documentation . . . . .	152
5.27.3.1 cleanUp() . . . . .	152
5.27.3.2 createFramebuffer() . . . . .	153
5.27.3.3 createGraphicsPipeline() . . . . .	154
5.27.3.4 createPushConstantRange() . . . . .	157
5.27.3.5 createRenderPass() . . . . .	157
5.27.3.6 createTextures() . . . . .	159
5.27.3.7 getOffscreenTexture() . . . . .	161
5.27.3.8 init() . . . . .	162
5.27.3.9 recordCommands() . . . . .	163
5.27.3.10 setPushConstant() . . . . .	165
5.27.3.11 shaderHotReload() . . . . .	166

---

5.27.4 Field Documentation . . . . .	166
5.27.4.1 commandBufferManager . . . . .	166
5.27.4.2 depthBufferImage . . . . .	167
5.27.4.3 device . . . . .	167
5.27.4.4 framebuffer . . . . .	167
5.27.4.5 graphics_pipeline . . . . .	167
5.27.4.6 offscreenTextures . . . . .	167
5.27.4.7 pipeline_layout . . . . .	168
5.27.4.8 push_constant_range . . . . .	168
5.27.4.9 pushConstant . . . . .	168
5.27.4.10 render_pass . . . . .	168
5.27.4.11 vulkanSwapChain . . . . .	168
5.28 Raytracing Class Reference . . . . .	169
5.28.1 Detailed Description . . . . .	170
5.28.2 Constructor & Destructor Documentation . . . . .	170
5.28.2.1 Raytracing() . . . . .	170
5.28.2.2 ~Raytracing() . . . . .	170
5.28.3 Member Function Documentation . . . . .	170
5.28.3.1 cleanUp() . . . . .	171
5.28.3.2 createGraphicsPipeline() . . . . .	171
5.28.3.3 createPCRange() . . . . .	174
5.28.3.4 createSBT() . . . . .	175
5.28.3.5 init() . . . . .	176
5.28.3.6 recordCommands() . . . . .	177
5.28.3.7 shaderHotReload() . . . . .	179
5.28.4 Field Documentation . . . . .	180
5.28.4.1 call_region . . . . .	180
5.28.4.2 device . . . . .	180
5.28.4.3 graphicsPipeline . . . . .	181
5.28.4.4 hit_region . . . . .	181
5.28.4.5 hitShaderBindingTableBuffer . . . . .	181
5.28.4.6 miss_region . . . . .	181
5.28.4.7 missShaderBindingTableBuffer . . . . .	181
5.28.4.8 pc . . . . .	182
5.28.4.9 pc_ranges . . . . .	182
5.28.4.10 pipeline_layout . . . . .	182
5.28.4.11 raygenShaderBindingTableBuffer . . . . .	182
5.28.4.12 raytracing_properties . . . . .	182
5.28.4.13 rgen_region . . . . .	183
5.28.4.14 shader_groups . . . . .	183
5.28.4.15 shaderBindingTableBuffer . . . . .	183
5.28.4.16 vulkanSwapChain . . . . .	183

---

5.29 Scene Class Reference . . . . .	184
5.29.1 Detailed Description . . . . .	185
5.29.2 Constructor & Destructor Documentation . . . . .	185
5.29.2.1 Scene() . . . . .	185
5.29.2.2 ~Scene() . . . . .	185
5.29.3 Member Function Documentation . . . . .	186
5.29.3.1 add_model() . . . . .	186
5.29.3.2 add_object_description() . . . . .	186
5.29.3.3 cleanUp() . . . . .	186
5.29.3.4 get_model_list() . . . . .	187
5.29.3.5 getGuiSceneSharedVars() . . . . .	187
5.29.3.6 getIndexBuffer() . . . . .	188
5.29.3.7 getIndexCount() . . . . .	188
5.29.3.8 getMeshCount() . . . . .	189
5.29.3.9 getModelCount() . . . . .	189
5.29.3.10 getModelMatrix() . . . . .	190
5.29.3.11 getNumberMeshes() . . . . .	190
5.29.3.12 getNumberObjectDescriptions() . . . . .	190
5.29.3.13 getObjectDescriptions() . . . . .	191
5.29.3.14 getTextureCount() . . . . .	191
5.29.3.15 getTextures() . . . . .	192
5.29.3.16 getTextureSampler() . . . . .	192
5.29.3.17 getVertexBuffer() . . . . .	193
5.29.3.18 loadModel() . . . . .	193
5.29.3.19 update_model_matrix() . . . . .	194
5.29.3.20 update_user_input() . . . . .	195
5.29.4 Field Documentation . . . . .	195
5.29.4.1 guiSceneSharedVars . . . . .	196
5.29.4.2 model_list . . . . .	196
5.29.4.3 object_descriptions . . . . .	196
5.30 SceneUBO Struct Reference . . . . .	196
5.30.1 Detailed Description . . . . .	197
5.30.2 Field Documentation . . . . .	197
5.30.2.1 cam_pos . . . . .	197
5.30.2.2 light_dir . . . . .	198
5.30.2.3 view_dir . . . . .	198
5.31 ShaderHelper Class Reference . . . . .	198
5.31.1 Detailed Description . . . . .	199
5.31.2 Constructor & Destructor Documentation . . . . .	199
5.31.2.1 ShaderHelper() . . . . .	199
5.31.2.2 ~ShaderHelper() . . . . .	199
5.31.3 Member Function Documentation . . . . .	199

5.31.3.1 compileShader()	200
5.31.3.2 createShaderModule()	201
5.31.3.3 getShaderSpvDir()	202
5.31.4 Field Documentation	202
5.31.4.1 target	202
5.32 PathTracing::SpecializationData Struct Reference	203
5.32.1 Detailed Description	203
5.32.2 Field Documentation	203
5.32.2.1 specWorkGroupSizeX	203
5.32.2.2 specWorkGroupSizeY	204
5.32.2.3 specWorkGroupSizeZ	204
5.33 SwapChainDetails Struct Reference	204
5.33.1 Detailed Description	205
5.33.2 Field Documentation	205
5.33.2.1 formats	205
5.33.2.2 presentation_mode	205
5.33.2.3 surface_capabilities	205
5.34 Texture Class Reference	206
5.34.1 Detailed Description	207
5.34.2 Constructor & Destructor Documentation	207
5.34.2.1 Texture()	207
5.34.2.2 ~Texture()	208
5.34.3 Member Function Documentation	208
5.34.3.1 cleanUp()	208
5.34.3.2 createFromFile()	209
5.34.3.3 createImage()	210
5.34.3.4 createImageView()	211
5.34.3.5 generateMipMaps()	212
5.34.3.6 getImage()	214
5.34.3.7 getImageView()	214
5.34.3.8 getMipLevel()	215
5.34.3.9 getVulkanImage()	215
5.34.3.10 getVulkanImageView()	216
5.34.3.11 loadTextureData()	216
5.34.3.12 setImage()	217
5.34.3.13 setImageView()	217
5.34.4 Field Documentation	218
5.34.4.1 commandBufferManager	218
5.34.4.2 mip_levels	218
5.34.4.3 vulkanBufferManager	218
5.34.4.4 vulkanImage	218
5.34.4.5 vulkanImageView	219

---

5.35 TopLevelAccelerationStructure Struct Reference . . . . .	219
5.35.1 Detailed Description . . . . .	220
5.35.2 Field Documentation . . . . .	220
5.35.2.1 vulkanAS . . . . .	220
5.35.2.2 vulkanBuffer . . . . .	220
5.36 Vertex Struct Reference . . . . .	221
5.36.1 Detailed Description . . . . .	222
5.36.2 Constructor & Destructor Documentation . . . . .	222
5.36.2.1 Vertex() [1/2] . . . . .	222
5.36.2.2 Vertex() [2/2] . . . . .	222
5.36.3 Member Function Documentation . . . . .	222
5.36.3.1 operator==() . . . . .	222
5.36.4 Field Documentation . . . . .	223
5.36.4.1 color [1/2] . . . . .	223
5.36.4.2 color [2/2] . . . . .	223
5.36.4.3 normal [1/2] . . . . .	223
5.36.4.4 normal [2/2] . . . . .	223
5.36.4.5 pos [1/2] . . . . .	223
5.36.4.6 pos [2/2] . . . . .	224
5.36.4.7 texture_coords [1/2] . . . . .	224
5.36.4.8 texture_coords [2/2] . . . . .	224
5.37 VulkanBuffer Class Reference . . . . .	224
5.37.1 Detailed Description . . . . .	226
5.37.2 Constructor & Destructor Documentation . . . . .	226
5.37.2.1 VulkanBuffer() . . . . .	226
5.37.2.2 ~VulkanBuffer() . . . . .	226
5.37.3 Member Function Documentation . . . . .	226
5.37.3.1 cleanUp() . . . . .	226
5.37.3.2 create() . . . . .	227
5.37.3.3 getBuffer() . . . . .	229
5.37.3.4 getBufferMemory() . . . . .	230
5.37.4 Field Documentation . . . . .	230
5.37.4.1 buffer . . . . .	230
5.37.4.2 bufferMemory . . . . .	230
5.37.4.3 created . . . . .	231
5.37.4.4 device . . . . .	231
5.38 VulkanBufferManager Class Reference . . . . .	231
5.38.1 Detailed Description . . . . .	233
5.38.2 Constructor & Destructor Documentation . . . . .	233
5.38.2.1 VulkanBufferManager() . . . . .	233
5.38.2.2 ~VulkanBufferManager() . . . . .	233
5.38.3 Member Function Documentation . . . . .	233

5.38.3.1 copyBuffer() . . . . .	234
5.38.3.2 copyImageBuffer() . . . . .	235
5.38.3.3 createBufferAndUploadVectorOnDevice() . . . . .	236
5.38.4 Field Documentation . . . . .	237
5.38.4.1 commandBufferManager . . . . .	238
5.39 VulkanDevice Class Reference . . . . .	238
5.39.1 Detailed Description . . . . .	239
5.39.2 Constructor & Destructor Documentation . . . . .	239
5.39.2.1 VulkanDevice() . . . . .	240
5.39.2.2 ~VulkanDevice() . . . . .	240
5.39.3 Member Function Documentation . . . . .	240
5.39.3.1 check_device_extension_support() . . . . .	241
5.39.3.2 check_device_suitable() . . . . .	241
5.39.3.3 cleanUp() . . . . .	242
5.39.3.4 create_logical_device() . . . . .	243
5.39.3.5 get_physical_device() . . . . .	245
5.39.3.6 getComputeQueue() . . . . .	246
5.39.3.7 getGraphicsQueue() . . . . .	246
5.39.3.8 getLogicalDevice() . . . . .	247
5.39.3.9 getPhysicalDevice() . . . . .	248
5.39.3.10 getPhysicalDeviceProperties() . . . . .	249
5.39.3.11 getPresentationQueue() . . . . .	249
5.39.3.12 getQueueFamilies() [1/2] . . . . .	250
5.39.3.13 getQueueFamilies() [2/2] . . . . .	251
5.39.3.14 getSwapchainDetails() [1/2] . . . . .	251
5.39.3.15 getSwapchainDetails() [2/2] . . . . .	252
5.39.4 Field Documentation . . . . .	253
5.39.4.1 compute_queue . . . . .	253
5.39.4.2 device_extensions . . . . .	253
5.39.4.3 device_extensions_for_raytracing . . . . .	253
5.39.4.4 device_properties . . . . .	254
5.39.4.5 graphics_queue . . . . .	254
5.39.4.6 instance . . . . .	254
5.39.4.7 logical_device . . . . .	254
5.39.4.8 physical_device . . . . .	254
5.39.4.9 presentation_queue . . . . .	255
5.39.4.10 surface . . . . .	255
5.40 VulkanImage Class Reference . . . . .	255
5.40.1 Detailed Description . . . . .	257
5.40.2 Constructor & Destructor Documentation . . . . .	257
5.40.2.1 VulkanImage() . . . . .	257
5.40.2.2 ~VulkanImage() . . . . .	257

---

5.40.3 Member Function Documentation . . . . .	257
5.40.3.1 accessFlagsForImageLayout() . . . . .	258
5.40.3.2 cleanUp() . . . . .	258
5.40.3.3 create() . . . . .	259
5.40.3.4 getImage() . . . . .	260
5.40.3.5 pipelineStageForLayout() . . . . .	261
5.40.3.6 setImage() . . . . .	262
5.40.3.7 transitionImageLayout() [1/2] . . . . .	262
5.40.3.8 transitionImageLayout() [2/2] . . . . .	263
5.40.4 Field Documentation . . . . .	265
5.40.4.1 commandBufferManager . . . . .	265
5.40.4.2 device . . . . .	265
5.40.4.3 image . . . . .	265
5.40.4.4 imageMemory . . . . .	265
5.41 VulkanImageView Class Reference . . . . .	266
5.41.1 Detailed Description . . . . .	267
5.41.2 Constructor & Destructor Documentation . . . . .	267
5.41.2.1 VulkanImageView() . . . . .	267
5.41.2.2 ~VulkanImageView() . . . . .	267
5.41.3 Member Function Documentation . . . . .	267
5.41.3.1 cleanUp() . . . . .	267
5.41.3.2 create() . . . . .	268
5.41.3.3 getImageView() . . . . .	269
5.41.3.4 setImageView() . . . . .	269
5.41.4 Field Documentation . . . . .	270
5.41.4.1 device . . . . .	270
5.41.4.2 imageView . . . . .	270
5.42 VulkanInstance Class Reference . . . . .	270
5.42.1 Detailed Description . . . . .	271
5.42.2 Constructor & Destructor Documentation . . . . .	272
5.42.2.1 VulkanInstance() . . . . .	272
5.42.2.2 ~VulkanInstance() . . . . .	273
5.42.3 Member Function Documentation . . . . .	273
5.42.3.1 check_instance_extension_support() . . . . .	273
5.42.3.2 check_validation_layer_support() . . . . .	274
5.42.3.3 cleanUp() . . . . .	275
5.42.3.4 getVulkanInstance() . . . . .	275
5.42.4 Field Documentation . . . . .	276
5.42.4.1 instance . . . . .	276
5.42.4.2 validationLayers . . . . .	276
5.43 VulkanRenderer Class Reference . . . . .	276
5.43.1 Detailed Description . . . . .	278

---

5.43.2 Constructor & Destructor Documentation . . . . .	278
5.43.2.1 VulkanRenderer() . . . . .	278
5.43.2.2 ~VulkanRenderer() . . . . .	280
5.43.3 Member Function Documentation . . . . .	281
5.43.3.1 checkChangedFrameBufferSize() . . . . .	281
5.43.3.2 cleanUp() . . . . .	283
5.43.3.3 cleanUpCommandPools() . . . . .	284
5.43.3.4 cleanUpSync() . . . . .	285
5.43.3.5 cleanUpUBOs() . . . . .	285
5.43.3.6 create_command_buffers() . . . . .	286
5.43.3.7 create_command_pool() . . . . .	287
5.43.3.8 create_object_description_buffer() . . . . .	288
5.43.3.9 create_post_descriptor_layout() . . . . .	289
5.43.3.10 create_surface() . . . . .	291
5.43.3.11 create_uniform_buffers() . . . . .	292
5.43.3.12 createDescriptorPoolSharedRenderStages() . . . . .	293
5.43.3.13 createRaytracingDescriptorPool() . . . . .	294
5.43.3.14 createRaytracingDescriptorSetLayouts() . . . . .	295
5.43.3.15 createRaytracingDescriptorSets() . . . . .	296
5.43.3.16 createSharedRenderDescriptorSet() . . . . .	297
5.43.3.17 createSharedRenderDescriptorSetLayouts() . . . . .	298
5.43.3.18 createSynchronization() . . . . .	299
5.43.3.19 drawFrame() . . . . .	300
5.43.3.20 finishAllRenderCommands() . . . . .	303
5.43.3.21 record_commands() . . . . .	304
5.43.3.22 shaderHotReload() . . . . .	306
5.43.3.23 update_raytracing_descriptor_set() . . . . .	307
5.43.3.24 update_uniform_buffers() . . . . .	308
5.43.3.25 updatePostDescriptorSets() . . . . .	309
5.43.3.26 updateRaytracingDescriptorSets() . . . . .	310
5.43.3.27 updateStateDueToUserInput() . . . . .	312
5.43.3.28 updateTexturesInSharedRenderDescriptorSet() . . . . .	313
5.43.3.29 updateUniforms() . . . . .	314
5.43.4 Field Documentation . . . . .	315
5.43.4.1 allocator . . . . .	316
5.43.4.2 asManager . . . . .	316
5.43.4.3 command_buffers . . . . .	316
5.43.4.4 commandBufferManager . . . . .	316
5.43.4.5 compute_command_pool . . . . .	316
5.43.4.6 current_frame . . . . .	317
5.43.4.7 descriptorPoolSharedRenderStages . . . . .	317
5.43.4.8 device . . . . .	317

---

5.43.4.9 globalUBO . . . . .	317
5.43.4.10 globalUBOBuffer . . . . .	318
5.43.4.11 graphics_command_pool . . . . .	318
5.43.4.12 gui . . . . .	318
5.43.4.13 image_available . . . . .	318
5.43.4.14 images_in_flight_fences . . . . .	318
5.43.4.15 in_flight_fences . . . . .	319
5.43.4.16 instance . . . . .	319
5.43.4.17 objectDescriptionBuffer . . . . .	319
5.43.4.18 pathTracing . . . . .	319
5.43.4.19 post_descriptor_pool . . . . .	319
5.43.4.20 post_descriptor_set . . . . .	320
5.43.4.21 post_descriptor_set_layout . . . . .	320
5.43.4.22 postStage . . . . .	320
5.43.4.23 rasterizer . . . . .	320
5.43.4.24 raytracingDescriptorPool . . . . .	320
5.43.4.25 raytracingDescriptorSet . . . . .	321
5.43.4.26 raytracingDescriptorsetLayout . . . . .	321
5.43.4.27 raytracingStage . . . . .	321
5.43.4.28 render_finished . . . . .	321
5.43.4.29 scene . . . . .	321
5.43.4.30 sceneUBO . . . . .	322
5.43.4.31 sceneUBOBuffer . . . . .	322
5.43.4.32 sharedRenderDescriptorSet . . . . .	322
5.43.4.33 sharedRenderDescriptorsetLayout . . . . .	322
5.43.4.34 surface . . . . .	322
5.43.4.35 vulkanBufferManager . . . . .	323
5.43.4.36 vulkanSwapChain . . . . .	323
5.43.4.37 window . . . . .	323
5.44 VulkanSwapChain Class Reference . . . . .	324
5.44.1 Detailed Description . . . . .	325
5.44.2 Constructor & Destructor Documentation . . . . .	325
5.44.2.1 VulkanSwapChain() . . . . .	325
5.44.2.2 ~VulkanSwapChain() . . . . .	325
5.44.3 Member Function Documentation . . . . .	325
5.44.3.1 choose_best_presentation_mode() . . . . .	326
5.44.3.2 choose_best_surface_format() . . . . .	326
5.44.3.3 choose_swap_extent() . . . . .	327
5.44.3.4 cleanUp() . . . . .	328
5.44.3.5 getNumberSwapChainImages() . . . . .	328
5.44.3.6 getSwapChain() . . . . .	329
5.44.3.7 getSwapChainExtent() . . . . .	330

---

5.44.3.8 getSwapChainFormat()	330
5.44.3.9 getSwapChainImage()	331
5.44.3.10 initVulkanContext()	331
5.44.4 Field Documentation	333
5.44.4.1 device	333
5.44.4.2 swap_chain_extent	334
5.44.4.3 swap_chain_image_format	334
5.44.4.4 swap_chain_images	334
5.44.4.5 swapchain	334
5.44.4.6 window	334
5.45 Window Class Reference	335
5.45.1 Detailed Description	336
5.45.2 Constructor & Destructor Documentation	336
5.45.2.1 Window() [1/2]	337
5.45.2.2 Window() [2/2]	337
5.45.2.3 ~Window()	338
5.45.3 Member Function Documentation	338
5.45.3.1 cleanUp()	338
5.45.3.2 framebuffer_size_callback()	338
5.45.3.3 framebuffer_size_has_changed()	339
5.45.3.4 get_buffer_height()	339
5.45.3.5 get_buffer_width()	340
5.45.3.6 get_height()	340
5.45.3.7 get_keys()	340
5.45.3.8 get_should_close()	340
5.45.3.9 get_width()	341
5.45.3.10 get_window()	341
5.45.3.11 get_x_change()	341
5.45.3.12 get_y_change()	342
5.45.3.13 init_callbacks()	342
5.45.3.14 initialize()	343
5.45.3.15 key_callback()	344
5.45.3.16 mouse_button_callback()	344
5.45.3.17 mouse_callback()	345
5.45.3.18 reset_framebuffer_has_changed()	346
5.45.3.19 set_buffer_size()	346
5.45.3.20 update_viewport()	347
5.45.4 Field Documentation	347
5.45.4.1 framebuffer_resized	347
5.45.4.2 keys	347
5.45.4.3 last_x	347
5.45.4.4 last_y	348

5.45.4.5 main_window . . . . .	348
5.45.4.6 mouse_first_moved . . . . .	348
5.45.4.7 window_buffer_height . . . . .	348
5.45.4.8 window_buffer_width . . . . .	348
5.45.4.9 window_height . . . . .	349
5.45.4.10 window_width . . . . .	349
5.45.4.11 x_change . . . . .	349
5.45.4.12 y_change . . . . .	349
<b>6 File Documentation</b>	<b>351</b>
6.1 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/CompilerWarnings.cmake File Reference	351
6.2 CompilerWarnings.cmake . . . . .	351
6.3 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/CompileShadersToSPV.cmake File Reference	352
6.4 CompileShadersToSPV.cmake . . . . .	352
6.5 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/Doxygen.cmake File Reference	353
6.6 Doxygen.cmake . . . . .	353
6.7 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetExternalLibsFilters.cmake File Reference	353
6.8 SetExternalLibsFilters.cmake . . . . .	353
6.9 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetProjectFilters.cmake File Reference	354
6.10 SetProjectFilters.cmake . . . . .	354
6.11 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetShaderFilters.cmake File Reference	356
6.12 SetShaderFilters.cmake . . . . .	356
6.13 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/Sanitizers.cmake File Reference	357
6.14 Sanitizers.cmake . . . . .	357
6.15 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/SetSourceGroups.cmake File Reference	357
6.16 SetSourceGroups.cmake . . . . .	357
6.17 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/StaticAnalyzers.cmake File Reference	358
6.18 StaticAnalyzers.cmake . . . . .	358
6.19 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/app/App.hpp File Reference	358
6.20 App.hpp . . . . .	358
6.21 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/FormatHelper.hpp File Reference	358
6.21.1 Function Documentation . . . . .	359
6.21.1.1 choose_supported_format() . . . . .	359
6.22 FormatHelper.hpp . . . . .	359
6.23 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/Globals.hpp File Reference	360
6.23.1 Variable Documentation . . . . .	360
6.23.1.1 MAX_FRAME_DRAWNS . . . . .	360
6.23.1.2 MAX_OBJECTS . . . . .	360
6.24 Globals.hpp . . . . .	360

---

6.25 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/MemoryHelper.hpp File Reference . . . . .	361
6.25.1 Function Documentation . . . . .	361
6.25.1.1 align_up() . . . . .	361
6.25.1.2 find_memory_type_index() . . . . .	361
6.26 MemoryHelper.hpp . . . . .	362
6.27 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/Utilities.hpp File Reference . . . . .	362
6.27.1 Variable Documentation . . . . .	363
6.27.1.1 ENABLE_VALIDATION_LAYERS . . . . .	363
6.28 Utilities.hpp . . . . .	363
6.29 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/gui/GUI.hpp File Reference . . . . .	363
6.30 GUI.hpp . . . . .	363
6.31 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/memory/Allocator.hpp File Reference . . . . .	364
6.32 Allocator.hpp . . . . .	364
6.33 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/← ASManager.hpp File Reference . . . . .	365
6.34 ASManager.hpp . . . . .	365
6.35 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/← BottomLevelAccelerationStructure.hpp File Reference . . . . .	366
6.36 BottomLevelAccelerationStructure.hpp . . . . .	366
6.37 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/← TopLevelAccelerationStructure.hpp File Reference . . . . .	366
6.38 TopLevelAccelerationStructure.hpp . . . . .	366
6.39 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/CommandBufferManager.hpp File Reference . . . . .	366
6.40 CommandBufferManager.hpp . . . . .	367
6.41 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/GlobalUBO.hpp File Reference . . . . .	367
6.41.1 Typedef Documentation . . . . .	367
6.41.1.1 mat4 . . . . .	367
6.41.1.2 uint . . . . .	367
6.41.1.3 vec2 . . . . .	368
6.41.1.4 vec3 . . . . .	368
6.41.1.5 vec4 . . . . .	368
6.42 GlobalUBO.hpp . . . . .	368
6.43 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/GUIRendererShared← Vars.hpp File Reference . . . . .	368
6.44 GUIRendererSharedVars.hpp . . . . .	369
6.45 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/PathTracing.hpp File Reference . . . . .	369
6.46 PathTracing.hpp . . . . .	369
6.47 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/PostStage.hpp File Reference . . . . .	370
6.48 PostStage.hpp . . . . .	370
6.49 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/Push← ConstantPathTracing.hpp File Reference . . . . .	371

---

6.49.1 Typedef Documentation . . . . .	371
6.49.1.1 mat4 . . . . .	371
6.49.1.2 uint . . . . .	371
6.49.1.3 vec2 . . . . .	371
6.49.1.4 vec3 . . . . .	372
6.49.1.5 vec4 . . . . .	372
6.50 PushConstantPathTracing.hpp . . . . .	372
6.51 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/PushConstantPost.hpp File Reference . . . . .	372
6.51.1 Typedef Documentation . . . . .	373
6.51.1.1 mat4 . . . . .	373
6.51.1.2 uint . . . . .	373
6.51.1.3 vec2 . . . . .	373
6.51.1.4 vec3 . . . . .	373
6.51.1.5 vec4 . . . . .	373
6.52 PushConstantPost.hpp . . . . .	374
6.53 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/PushConstantRasterizer.hpp File Reference . . . . .	374
6.53.1 Typedef Documentation . . . . .	374
6.53.1.1 mat4 . . . . .	374
6.53.1.2 uint . . . . .	375
6.53.1.3 vec2 . . . . .	375
6.53.1.4 vec3 . . . . .	375
6.53.1.5 vec4 . . . . .	375
6.54 PushConstantRasterizer.hpp . . . . .	375
6.55 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/PushConstantRayTracing.hpp File Reference . . . . .	376
6.55.1 Typedef Documentation . . . . .	376
6.55.1.1 mat4 . . . . .	376
6.55.1.2 uint . . . . .	376
6.55.1.3 vec2 . . . . .	376
6.55.1.4 vec3 . . . . .	377
6.55.1.5 vec4 . . . . .	377
6.56 PushConstantRayTracing.hpp . . . . .	377
6.57 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/QueueFamilyIndices.hpp File Reference . . . . .	377
6.58 QueueFamilyIndices.hpp . . . . .	377
6.59 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/Rasterizer.hpp File Reference	378
6.60 Rasterizer.hpp . . . . .	378
6.61 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/Raytracing.hpp File Reference	378
6.62 Raytracing.hpp . . . . .	379
6.63 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/SceneUBO.hpp File Reference	379
6.63.1 Typedef Documentation . . . . .	380

---

6.63.1.1 mat4 . . . . .	380
6.63.1.2 uint . . . . .	380
6.63.1.3 vec2 . . . . .	380
6.63.1.4 vec3 . . . . .	380
6.63.1.5 vec4 . . . . .	380
6.64 SceneUBO.hpp . . . . .	381
6.65 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/SwapChainDetails.hpp File Reference . . . . .	381
6.66 SwapChainDetails.hpp . . . . .	381
6.67 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/VulkanRenderer.hpp File Reference . . . . .	381
6.68 VulkanRenderer.hpp . . . . .	382
6.69 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/VulkanRendererConfig.h File Reference . . . . .	383
6.70 VulkanRendererConfig.h . . . . .	383
6.71 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/VulkanRendererConfig.hpp File Reference . . . . .	384
6.72 VulkanRendererConfig.hpp . . . . .	384
6.73 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Camera.hpp File Reference . . . . .	384
6.74 Camera.hpp . . . . .	384
6.75 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/GUISceneSharedVars.hpp File Reference . . . . .	385
6.76 GUISceneSharedVars.hpp . . . . .	385
6.77 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Mesh.hpp File Reference . . . . .	385
6.78 Mesh.hpp . . . . .	386
6.79 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Model.hpp File Reference . . . . .	386
6.80 Model.hpp . . . . .	387
6.81 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ObjectDescription.hpp File Reference . . . . .	387
6.82 ObjectDescription.hpp . . . . .	387
6.83 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ObjLoader.hpp File Reference . . . . .	388
6.84 ObjLoader.hpp . . . . .	388
6.85 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ObjMaterial.hpp File Reference . . . . .	388
6.85.1 Typedef Documentation . . . . .	389
6.85.1.1 mat4 . . . . .	389
6.85.1.2 uint . . . . .	389
6.85.1.3 vec2 . . . . .	389
6.85.1.4 vec3 . . . . .	389
6.85.1.5 vec4 . . . . .	389
6.86 ObjMaterial.hpp . . . . .	390
6.87 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Scene.hpp File Reference . . . . .	390
6.88 Scene.hpp . . . . .	390
6.89 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/SceneConfig.hpp File Reference . . . . .	391
6.90 SceneConfig.hpp . . . . .	392

---

6.91 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Texture.hpp File Reference . . . . .	392
6.92 Texture.hpp . . . . .	392
6.93 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Vertex.hpp File Reference . . . . .	393
6.94 Vertex.hpp . . . . .	393
6.95 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/util/File.hpp File Reference . . . . .	394
6.96 File.hpp . . . . .	394
6.97 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ShaderHelper.hpp File Reference . . . . .	394
6.98 ShaderHelper.hpp . . . . .	395
6.99 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanBuffer.hpp File Reference . . . . .	395
6.100 VulkanBuffer.hpp . . . . .	395
6.101 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanBufferManager.hpp File Reference . . . . .	396
6.102 VulkanBufferManager.hpp . . . . .	396
6.103 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanDebug.hpp File Reference . . . . .	397
6.104 VulkanDebug.hpp . . . . .	397
6.105 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanDevice.hpp File Reference . . . . .	398
6.106 VulkanDevice.hpp . . . . .	398
6.107 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanImage.hpp File Reference . . . . .	399
6.108 VulkanImage.hpp . . . . .	399
6.109 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanImageView.hpp File Reference . . . . .	399
6.110 VulkanImageView.hpp . . . . .	400
6.111 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanInstance.hpp File Reference . . . . .	400
6.112 VulkanInstance.hpp . . . . .	400
6.113 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/VulkanSwapChain.hpp File Reference . . . . .	401
6.114 VulkanSwapChain.hpp . . . . .	401
6.115 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/window/Window.hpp File Reference . . . . .	401
6.116 Window.hpp . . . . .	402
6.117 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/disney.glsl File Reference . . . . .	402
6.118 disney.glsl . . . . .	402
6.119 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/frostbite.glsl File Reference . . . . .	405
6.120 frostbite.glsl . . . . .	405
6.121 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/pbrBook.glsl File Reference . . . . .	406
6.122 pbrBook.glsl . . . . .	406
6.123 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/phong.glsl File Reference . . . . .	407

---

6.124 phong.glsl . . . . .	407
6.125 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/unreal4.glsl File Reference . . . . .	408
6.126 unreal4.glsl . . . . .	408
6.127 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/Matlib.glsl File Reference . . . . .	409
6.128 Matlib.glsl . . . . .	409
6.129 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/microfacet.glsl File Reference . . . . .	410
6.130 microfacet.glsl . . . . .	410
6.131 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/raycommon.glsl File Reference . . . . .	410
6.132 raycommon.glsl . . . . .	410
6.133 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/Shading← Library.glsl File Reference . . . . .	410
6.134 ShadingLibrary.glsl . . . . .	410
6.135 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/hostDevice/host_← device_shared_vars.hpp File Reference . . . . .	411
6.135.1 Variable Documentation . . . . .	411
6.135.1.1 MAX_TEXTURE_COUNT . . . . .	412
6.136 host_device_shared_vars.hpp . . . . .	412
6.137 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/path_tracing/path_← tracing.comp File Reference . . . . .	412
6.138 path_tracing.comp . . . . .	412
6.139 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/post/post.frag File Reference . . . . .	415
6.140 post.frag . . . . .	415
6.141 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/post/post.frag.log.txt File Reference . . . . .	416
6.142 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/post/post.vert File Reference . . . . .	416
6.143 post.vert . . . . .	416
6.144 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/post/post.vert.log.txt File Reference . . . . .	416
6.145 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/rasterizer/shader.frag File Reference . . . . .	416
6.146 shader.frag . . . . .	416
6.147 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/rasterizer/shader.frag.← log.txt File Reference . . . . .	418
6.148 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/rasterizer/shader.vert File Reference . . . . .	418
6.149 shader.vert . . . . .	418
6.150 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/rasterizer/shader.vert.← log.txt File Reference . . . . .	419
6.151 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rchit File Reference . . . . .	419

---

6.152 raytrace.rchit . . . . .	419
6.153 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rchit.← log.txt File Reference . . . . .	421
6.154 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rgen File Reference . . . . .	421
6.155 raytrace.rgen . . . . .	421
6.156 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rgen.← log.txt File Reference . . . . .	422
6.157 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rmiss File Reference . . . . .	422
6.158 raytrace.rmiss . . . . .	422
6.159 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rmiss.← log.txt File Reference . . . . .	423
6.160 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/shadow.rmiss File Reference . . . . .	423
6.161 shadow.rmiss . . . . .	423
6.162 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/shadow.rmiss.← log.txt File Reference . . . . .	423
6.163 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/app/App.cpp File Reference . . . . .	423
6.164 App.cpp . . . . .	423
6.165 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/gui/GUI.cpp File Reference . . . . .	424
6.166 GUI.cpp . . . . .	424
6.167 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/Main.cpp File Reference . . . . .	427
6.167.1 Function Documentation . . . . .	427
6.167.1.1 main() . . . . .	427
6.168 Main.cpp . . . . .	428
6.169 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/memory/Allocator.cpp File Reference . . . . .	428
6.170 Allocator.cpp . . . . .	428
6.171 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/accelerationStructures/← ASManager.cpp File Reference . . . . .	428
6.172 ASManager.cpp . . . . .	428
6.173 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/CommandBufferManager.cpp File Reference . . . . .	435
6.174 CommandBufferManager.cpp . . . . .	435
6.175 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/PathTracing.cpp File Reference . . . . .	436
6.176 PathTracing.cpp . . . . .	436
6.177 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/PostStage.cpp File Reference . . . . .	439
6.178 PostStage.cpp . . . . .	439
6.179 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/Rasterizer.cpp File Reference . . . . .	445
6.180 Rasterizer.cpp . . . . .	445
6.181 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/Raytracing.cpp File Reference . . . . .	452
6.182 Raytracing.cpp . . . . .	452
6.183 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/VulkanRenderer.cpp File Ref- erence . . . . .	456
6.184 VulkanRenderer.cpp . . . . .	456

---

6.185 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/Camera.cpp File Reference . . . . .	470
6.186 Camera.cpp . . . . .	470
6.187 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/Mesh.cpp File Reference . . . . .	472
6.188 Mesh.cpp . . . . .	472
6.189 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/Model.cpp File Reference . . . . .	473
6.190 Model.cpp . . . . .	473
6.191 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ObjLoader.cpp File Reference . . . . .	475
6.192 ObjLoader.cpp . . . . .	475
6.193 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/Scene.cpp File Reference . . . . .	477
6.194 Scene.cpp . . . . .	477
6.195 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/SceneConfig.cpp File Reference . . . . .	478
6.196 SceneConfig.cpp . . . . .	478
6.197 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/Texture.cpp File Reference . . . . .	479
6.198 Texture.cpp . . . . .	479
6.199 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/Vertex.cpp File Reference . . . . .	482
6.200 Vertex.cpp . . . . .	482
6.201 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/util/File.cpp File Reference . . . . .	483
6.202 File.cpp . . . . .	483
6.203 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ShaderHelper.cpp File Reference . . . . .	484
6.204 ShaderHelper.cpp . . . . .	484
6.205 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanBuffer.cpp File Reference . . . . .	485
6.206 VulkanBuffer.cpp . . . . .	485
6.207 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanBufferManager.cpp File Reference . . . . .	486
6.208 VulkanBufferManager.cpp . . . . .	486
6.209 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanDebug.cpp File Reference . . . . .	487
6.210 VulkanDebug.cpp . . . . .	487
6.211 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanDevice.cpp File Reference . . . . .	488
6.212 VulkanDevice.cpp . . . . .	488
6.213 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanImage.cpp File Reference . . . . .	493
6.214 VulkanImage.cpp . . . . .	493
6.215 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanImageView.cpp File Reference . . . . .	496
6.216 VulkanImageView.cpp . . . . .	496
6.217 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanInstance.cpp File Reference . . . . .	496
6.218 VulkanInstance.cpp . . . . .	496
6.219 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanSwapChain.cpp File Reference . . . . .	498
6.220 VulkanSwapChain.cpp . . . . .	498

---

6.221 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/window/Window.cpp File Reference . . .	501
6.221.1 Function Documentation . . . . .	501
6.221.1.1 onErrorCallback() . . . . .	501
6.222 Window.cpp . . . . .	501
6.223 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/commit/cmake/SetTestFilters.cmake File Reference . . . . .	504
6.224 SetTestFilters.cmake . . . . .	504
6.225 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/compile/cmake/SetTestFilters.cmake File Reference . . . . .	504
6.226 SetTestFilters.cmake . . . . .	504
6.227 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/commit/CMakeLists.txt File Reference . . . . .	504
6.227.1 Function Documentation . . . . .	504
6.227.1.1 include() . . . . .	504
6.227.1.2 set() [1/2] . . . . .	505
6.227.1.3 set() [2/2] . . . . .	505
6.228 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/compile/CMakeLists.txt File Reference . . . . .	505
6.228.1 Function Documentation . . . . .	505
6.228.1.1 include() . . . . .	505
6.228.1.2 set() [1/2] . . . . .	506
6.228.1.3 set() [2/2] . . . . .	506
6.229 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/commit/commitSuite.cpp File Reference . . . . .	506
6.229.1 Function Documentation . . . . .	506
6.229.1.1 TEST() [1/2] . . . . .	506
6.229.1.2 TEST() [2/2] . . . . .	507
6.230 commitSuite.cpp . . . . .	507
6.231 C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/compile/compileSuite.cpp File Reference . . . . .	507
6.231.1 Function Documentation . . . . .	508
6.231.1.1 TEST() [1/2] . . . . .	508
6.231.1.2 TEST() [2/2] . . . . .	508
6.232 compileSuite.cpp . . . . .	508



# Chapter 1

## Namespace Index

### 1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

debug . . . . .	9
sceneConfig . . . . .	13
std . . . . .	15
vertex . . . . .	15



# Chapter 2

## Data Structure Index

### 2.1 Data Structures

Here are the data structures with brief descriptions:

Allocator	17
App	20
ASManager	22
BlasInput	37
BottomLevelAccelerationStructure	39
BuildAccelerationStructure	40
Camera	43
CommandBufferManager	54
File	57
GlobalUBO	60
GUI	62
GUIRendererSharedVars	72
GUISceneSharedVars	74
std::hash< Vertex >	75
Mesh	76
Model	91
ObjectDescription	100
ObjLoader	102
ObjMaterial	110
PathTracing	112
PostStage	124
PushConstantPathTracing	143
PushConstantPost	145
PushConstantRasterizer	146
PushConstantRaytracing	147
QueueFamilyIndices	148
Rasterizer	150
Raytracing	169
Scene	184
SceneUBO	196
ShaderHelper	198
PathTracing::SpecializationData	203
SwapChainDetails	204
Texture	206
TopLevelAccelerationStructure	219

<a href="#">Vertex</a>	221
<a href="#">VulkanBuffer</a>	224
<a href="#">VulkanBufferManager</a>	231
<a href="#">VulkanDevice</a>	238
<a href="#">VulkanImage</a>	255
<a href="#">VulkanImageView</a>	266
<a href="#">VulkanInstance</a>	270
<a href="#">VulkanRenderer</a>	276
<a href="#">VulkanSwapChain</a>	324
<a href="#">Window</a>	335

# Chapter 3

## File Index

### 3.1 File List

Here is a list of all files with brief descriptions:

C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/CompilerWarnings.cmake . . . . .	351
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/CompileShadersToSPV.cmake . . . . .	352
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/Doxygen.cmake . . . . .	353
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/Sanitizers.cmake . . . . .	357
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/SetSourceGroups.cmake . . . . .	357
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/StaticAnalyzers.cmake . . . . .	358
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetExternalLibsFilters.cmake . . . . .	353
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetProjectFilters.cmake . . . . .	354
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetShaderFilters.cmake . . . . .	356
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/app/App.hpp . . . . .	358
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/FormatHelper.hpp . . . . .	358
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/Globals.hpp . . . . .	360
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/MemoryHelper.hpp . . . . .	361
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/common/Utilities.hpp . . . . .	362
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/gui/GUI.hpp . . . . .	363
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/memory/Allocator.hpp . . . . .	364
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/CommandBufferManager.hpp . . . . .	366
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/GlobalUBO.hpp . . . . .	367
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/GUIRendererSharedVars.hpp . . . . .	368
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/PathTracing.hpp . . . . .	369
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/PostStage.hpp . . . . .	370
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/QueueFamilyIndices.hpp . . . . .	377
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/Rasterizer.hpp . . . . .	378
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/Raytracing.hpp . . . . .	378
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/SceneUBO.hpp . . . . .	379
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/SwapChainDetails.hpp . . . . .	381
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/VulkanRenderer.hpp . . . . .	381
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/VulkanRendererConfig.h . . . . .	383
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/VulkanRendererConfig.hpp . . . . .	384
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/ASManager.hpp 365	
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/BottomLevelAccelerationStructure 366	
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/TopLevelAccelerationStructure.hpp 366	

C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/ <a href="#">PushConstantPathTracing.hpp</a>	371
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/ <a href="#">PushConstantPost.hpp</a>	372
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/ <a href="#">PushConstantRasterizer.hpp</a>	374
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/ <a href="#">PushConstantRayTracing.hpp</a>	376
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">Camera.hpp</a>	384
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">GUISceneSharedVars.hpp</a>	385
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">Mesh.hpp</a>	385
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">Model.hpp</a>	386
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">ObjectDescription.hpp</a>	387
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">ObjLoader.hpp</a>	388
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">ObjMaterial.hpp</a>	388
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">Scene.hpp</a>	390
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">SceneConfig.hpp</a>	391
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">Texture.hpp</a>	392
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ <a href="#">Vertex.hpp</a>	393
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/util/ <a href="#">File.hpp</a>	394
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">ShaderHelper.hpp</a>	394
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanBuffer.hpp</a>	395
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanBufferManager.hpp</a>	396
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanDebug.hpp</a>	397
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanDevice.hpp</a>	398
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanImage.hpp</a>	399
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanImageView.hpp</a>	399
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanInstance.hpp</a>	400
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan_base/ <a href="#">VulkanSwapChain.hpp</a>	401
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/window/ <a href="#">Window.hpp</a>	401
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/ <a href="#">disney.glsl</a>	402
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/ <a href="#">frostbite.glsl</a>	405
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/ <a href="#">pbrBook.glsl</a>	406
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/ <a href="#">phong.glsl</a>	407
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/brdf/ <a href="#">unreal4.glsl</a>	408
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/ <a href="#">Matlib.glsl</a>	409
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/ <a href="#">microfacet.glsl</a>	410
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/ <a href="#">raycommon.glsl</a>	410
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/common/ <a href="#">ShadingLibrary.glsl</a>	410
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/hostDevice/ <a href="#">host_device_shared_vars.hpp</a>	411
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/path_tracing/ <a href="#">path_tracing.comp</a>	412
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/post/ <a href="#">post.frag</a>	415
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/post/ <a href="#">post.vert</a>	416
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/rasterizer/ <a href="#">shader.frag</a>	416
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/rasterizer/ <a href="#">shader.vert</a>	418
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/ <a href="#">raytrace.rchit</a>	419
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/ <a href="#">raytrace.agen</a>	421
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/ <a href="#">raytrace.rmiss</a>	422
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/ <a href="#">shadow.rmiss</a>	423
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/ <a href="#">Main.cpp</a>	427
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/app/ <a href="#">App.cpp</a>	423
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/gui/ <a href="#">GUI.cpp</a>	424
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/memory/ <a href="#">Allocator.cpp</a>	428
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/ <a href="#">CommandBufferManager.cpp</a>	435
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/ <a href="#">PathTracing.cpp</a>	436
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/ <a href="#">PostStage.cpp</a>	439

C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/ <a href="#">Rasterizer.cpp</a>	445
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/ <a href="#">Raytracing.cpp</a>	452
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/ <a href="#">VulkanRenderer.cpp</a>	456
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/accelerationStructures/ <a href="#">ASManager.cpp</a>	
428	
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">Camera.cpp</a>	470
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">Mesh.cpp</a>	472
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">Model.cpp</a>	473
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">ObjLoader.cpp</a>	475
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">Scene.cpp</a>	477
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">SceneConfig.cpp</a>	478
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">Texture.cpp</a>	479
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ <a href="#">Vertex.cpp</a>	482
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/util/ <a href="#">File.cpp</a>	483
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">ShaderHelper.cpp</a>	484
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanBuffer.cpp</a>	485
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanBufferManager.cpp</a>	486
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanDebug.cpp</a>	487
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanDevice.cpp</a>	488
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanImage.cpp</a>	493
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanImageView.cpp</a>	496
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanInstance.cpp</a>	496
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan_base/ <a href="#">VulkanSwapChain.cpp</a>	498
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Src/window/ <a href="#">Window.cpp</a>	501
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/commit/ <a href="#">commitSuite.cpp</a>	506
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/commit/cmake/ <a href="#">SetTestFilters.cmake</a>	504
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/compile/ <a href="#">compileSuite.cpp</a>	507
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Test/compile/cmake/ <a href="#">SetTestFilters.cmake</a>	504



# Chapter 4

## Namespace Documentation

### 4.1 debug Namespace Reference

#### Functions

- VKAPI\_ATTR VkBool32 VKAPI\_CALL [messageCallback](#) (VkDebugReportFlagsEXT flags, VkDebugReportObjectTypesEXT objType, uint64\_t srcObject, size\_t location, int32\_t msgCode, const char \*pLayerPrefix, const char \*pMsg, void \*pUserData)
- void [setupDebugging](#) (VkInstance instance, VkDebugReportFlagsEXT flags, VkDebugReportCallbackEXT callBack)
- void [freeDebugCallback](#) (VkInstance instance)
- VKAPI\_ATTR VkBool32 VKAPI\_CALL [debugUtilsMessengerCallback](#) (VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity, VkDebugUtilsMessageTypeFlagsEXT messageType, const VkDebugUtilsMessengerCallbackDataEXT \*pCallbackData, void \*pUserData)

#### Variables

- int [validationLayerCount](#)
- const char \* [validationLayerNames](#) []
- PFN\_vkCreateDebugUtilsMessengerEXT [vkCreateDebugUtilsMessengerEXT](#)
- PFN\_vkDestroyDebugUtilsMessengerEXT [vkDestroyDebugUtilsMessengerEXT](#)
- VkDebugUtilsMessengerEXT [debugUtilsMessenger](#)

#### 4.1.1 Function Documentation

#### 4.1.1.1 debugUtilsMessengerCallback()

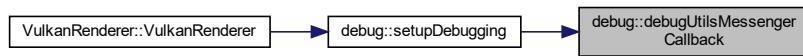
```
VKAPI_ATTR VkBool32 VKAPI_CALL debug::debugUtilsMessengerCallback (
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageType,
    const VkDebugUtilsMessengerCallbackDataEXT * pCallbackData,
    void * pUserData )
```

Definition at line 10 of file [VulkanDebug.cpp](#).

```
00014     {
00015     // Select prefix depending on flags passed to the callback
00016     std::string prefix("");
00017
00018     if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT) {
00019         prefix = "VERBOSE: ";
00020     } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT) {
00021         prefix = "INFO: ";
00022     } else if (messageSeverity &
00023                 VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
00024         prefix = "WARNING: ";
00025     } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
00026         prefix = "ERROR: ";
00027     }
00028
00029     // Display message to default output (console/logcat)
00030     std::stringstream debugMessage;
00031     debugMessage << prefix << "[" << pCallbackData->messageIdNumber << "] ["
00032             << pCallbackData->pMessageIdName
00033             << "] : " << pCallbackData->pMessage;
00034
00035 #if defined(__ANDROID__)
00036     if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
00037         LOGE("%s", debugMessage.str().c_str());
00038     } else {
00039         LOGD("%s", debugMessage.str().c_str());
00040     }
00041 #else
00042     if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
00043         std::cerr << debugMessage.str() << "\n";
00044     } else {
00045         std::cout << debugMessage.str() << "\n";
00046     }
00047     fflush(stdout);
00048 #endif
00049
00050     // The return value of this callback controls whether the Vulkan call that
00051     // caused the validation message will be aborted or not. We return VK_FALSE as
00052     // we DON'T want Vulkan calls that cause a validation message to abort. If you
00053     // instead want to have calls abort, pass in VK_TRUE and the function will
00054     // return VK_ERROR_VALIDATION_FAILED_EXT
00055     return VK_FALSE;
00056 }
```

Referenced by [setupDebugging\(\)](#).

Here is the caller graph for this function:



### 4.1.1.2 freeDebugCallback()

```
void debug::freeDebugCallback (
    VkInstance instance )
```

Definition at line 82 of file [VulkanDebug.cpp](#).

```
00082     {
00083     if (debugUtilsMessenger != VK_NULL_HANDLE) {
00084         vkDestroyDebugUtilsMessengerEXT(instance, debugUtilsMessenger, nullptr);
00085     }
00086 }
```

References [debugUtilsMessenger](#), and [vkDestroyDebugUtilsMessengerEXT](#).

Referenced by [VulkanRenderer::cleanUp\(\)](#).

Here is the caller graph for this function:



### 4.1.1.3 messageCallback()

```
VKAPI_ATTR VkBool32 VKAPI_CALL debug::messageCallback (
    VkDebugReportFlagsEXT flags,
    VkDebugReportObjectTypeEXT objType,
    uint64_t srcObject,
    size_t location,
    int32_t msgCode,
    const char * pLayerPrefix,
    const char * pMsg,
    void * pUserData )
```

### 4.1.1.4 setupDebugging()

```
void debug::setupDebugging (
    VkInstance instance,
    VkDebugReportFlagsEXT flags,
    VkDebugReportCallbackEXT callBack )
```

Definition at line 58 of file [VulkanDebug.cpp](#).

```
00059     {
00060     vkCreateDebugUtilsMessengerEXT =
00061         reinterpret_cast<PFN_vkCreateDebugUtilsMessengerEXT>(
00062             vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT"));
00063     vkDestroyDebugUtilsMessengerEXT =
```

```

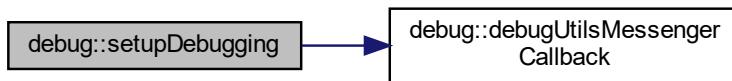
00064     reinterpret_cast<PFN_vkDestroyDebugUtilsMessengerEXT>(
00065         vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT"));
00066
00067     VkDebugUtilsMessengerCreateInfoEXT debugUtilsMessengerCI{};
00068     debugUtilsMessengerCI.sType =
00069         VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
00070     debugUtilsMessengerCI.messageSeverity =
00071         VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
00072         VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
00073     debugUtilsMessengerCI.messageType =
00074         VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
00075         VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT;
00076     debugUtilsMessengerCI.pfnUserCallback = debugUtilsMessengerCallback;
00077     ASSERT_VULKAN(vkCreateDebugUtilsMessengerEXT(instance, &debugUtilsMessengerCI,
00078                                                 nullptr, &debugUtilsMessenger),
00079                 "Failed to create debug messenger")
00080 }

```

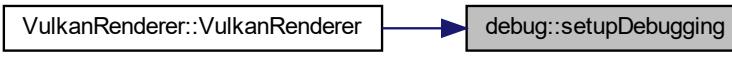
References [debugUtilsMessenger](#), [debugUtilsMessengerCallback\(\)](#), [vkCreateDebugUtilsMessengerEXT](#), and [vkDestroyDebugUtilsMessengerEXT](#).

Referenced by [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



## 4.1.2 Variable Documentation

### 4.1.2.1 debugUtilsMessenger

VkDebugUtilsMessengerEXT debug::debugUtilsMessenger

Definition at line 8 of file [VulkanDebug.cpp](#).

Referenced by [freeDebugCallback\(\)](#), and [setupDebugging\(\)](#).

#### 4.1.2.2 validationLayerCount

```
int debug::validationLayerCount [extern]
```

#### 4.1.2.3 validationLayerNames

```
const char* debug::validationLayerNames[] [extern]
```

#### 4.1.2.4 vkCreateDebugUtilsMessengerEXT

```
PFN_vkCreateDebugUtilsMessengerEXT debug::vkCreateDebugUtilsMessengerEXT
```

Definition at line [6](#) of file [VulkanDebug.cpp](#).

Referenced by [setupDebugging\(\)](#).

#### 4.1.2.5 vkDestroyDebugUtilsMessengerEXT

```
PFN_vkDestroyDebugUtilsMessengerEXT debug::vkDestroyDebugUtilsMessengerEXT
```

Definition at line [7](#) of file [VulkanDebug.cpp](#).

Referenced by [freeDebugCallback\(\)](#), and [setupDebugging\(\)](#).

## 4.2 sceneConfig Namespace Reference

### Functions

- std::string [getModelFile\(\)](#)
- [glm::mat4 getModelMatrix\(\)](#)

#### 4.2.1 Function Documentation

#### 4.2.1.1 getModelFile()

```
std::string sceneConfig::getModelFile ( )
```

Definition at line 9 of file SceneConfig.cpp.

```
00009     {
00010     std::stringstream modelFile;
00011     std::filesystem::path cwd = std::filesystem::current_path();
00012     modelFile << cwd.string();
00013     modelFile << RELATIVE_RESOURCE_PATH;
00014
00015 #if NDEBUG
00016     modelFile << "Models/crytek-sponza/";
00017     modelFile << "sponza_triang.obj";
00018
00019 #else
00020 #ifdef SULO_MODE
00021     modelFile << "Model/Sulo/WolfStahl/";
00022     //modelFile << "Wolf-Stahl.obj";
00023     modelFile << "SuloLongDongLampe_v2.obj";
00024 #else
00025     modelFile << "Models/VikingRoom/";
00026     modelFile << "viking_room.obj";
00027 #endif
00028 #endif
00029
00030     return modelFile.str();
00031 // std::string modelFile =
00032 // "Models/crytek-sponza/sponza_triang.obj"; std::string modelFile
00033 // = "Models/Dinosaurs/dinosaurs.obj"; std::string modelFile =
00034 // "Models/Pillum/PillumPainting_export.obj"; std::string modelFile
00035 // = "Models/sibenik/sibenik.obj"; std::string modelFile =
00036 // "Models/sportsCar/sportsCar.obj"; std::string modelFile =
00037 // "Models/StanfordDragon/dragon.obj"; std::string modelFile =
00038 // "Models/CornellBox/CornellBox-Sphere.obj"; std::string
00039 // "Models/bunny/bunny.obj"; std::string modelFile =
00040 // "Models/buddha/buddha.obj"; std::string modelFile =
00041 // "Models/bmw/bmw.obj"; std::string modelFile =
00042 // "Models/testScene.obj"; std::string modelFile =
00043 // "Models/San_Miguel/san-miguel-low-poly.obj";
00044 }
```

Referenced by [Scene::loadModel\(\)](#).

Here is the caller graph for this function:



#### 4.2.1.2 getModelMatrix()

```
glm::mat4 sceneConfig::getModelMatrix ( )
```

Definition at line 46 of file SceneConfig.cpp.

```
00046     {
00047     glm::mat4 modelMatrix(1.0f);
00048
00049 #if NDEBUG
00050
00051     // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00052     // -50.0f));
00053     modelMatrix = glm::scale(modelMatrix, glm::vec3(1.0f, 1.0f, 1.0f));
00054     /*dragon_model = glm::rotate(dragon_model, glm::radians(-90.f),
00055     glm::vec3(1.0f, 0.0f, 0.0f)); dragon_model = glm::rotate(dragon_model,
```

```
00056     glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));*/
00057
00058 #else
00059
00060 // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00061 // -50.0f));
00062 #if SULO_MODE
00063     modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00064 #else
00065     modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00066     modelMatrix = glm::rotate(modelMatrix, glm::radians(-90.f),
00067                               glm::vec3(1.0f, 0.0f, 0.0f));
00068     modelMatrix =
00069         glm::rotate(modelMatrix, glm::radians(90.f), glm::vec3(0.0f, 0.0f, 1.0f));
00070 #endif
00071
00072 #endif
00073
00074     return modelMatrix;
00075 }
```

Referenced by [Scene::loadModel\(\)](#).

Here is the caller graph for this function:



## 4.3 std Namespace Reference

### Data Structures

- struct [hash< Vertex >](#)

## 4.4 vertex Namespace Reference

### Functions

- std::array< VkVertexInputAttributeDescription, 4 > [getVertexInputAttributeDesc \(\)](#)

#### 4.4.1 Function Documentation

#### 4.4.1.1 `getVertexInputAttributeDesc()`

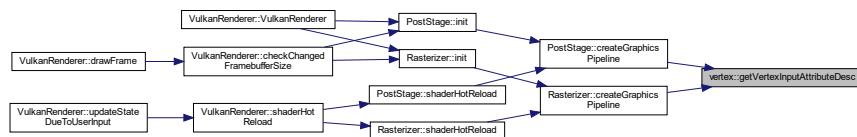
```
std::array< VkVertexInputAttributeDescription, 4 > vertex::getVertexInputAttributeDesc ( )
```

Definition at line 20 of file `Vertex.cpp`.

```
00020
00021     std::array<VkVertexInputAttributeDescription, 4> attribute_descriptions;
00022
00023     // Position attribute
00024     attribute_descriptions[0].binding = 0;
00025     attribute_descriptions[0].location = 0;
00026     attribute_descriptions[0].format =
00027         VK_FORMAT_R32G32B32_SFLOAT; // format data will take (also helps define
00028                                // size of data)
00029     attribute_descriptions[0].offset = offsetof(Vertex, pos);
00030
00031     // normal coord attribute
00032     attribute_descriptions[1].binding = 0;
00033     attribute_descriptions[1].location = 1;
00034     attribute_descriptions[1].format =
00035         VK_FORMAT_R32G32B32_SFLOAT; // format data will take (also helps define
00036                                // size of data)
00037     attribute_descriptions[1].offset =
00038         offsetof(Vertex, normal); // where this attribute is defined in the data
00039                                // for a single vertex
00040
00041     // normal coord attribute
00042     attribute_descriptions[2].binding = 0;
00043     attribute_descriptions[2].location = 2;
00044     attribute_descriptions[2].format =
00045         VK_FORMAT_R32G32B32_SFLOAT; // format data will take (also helps define
00046                                // size of data)
00047     attribute_descriptions[2].offset = offsetof(Vertex, color);
00048
00049     attribute_descriptions[3].binding = 0;
00050     // texture coord attribute
00051     attribute_descriptions[3].location = 3;
00052     attribute_descriptions[3].format =
00053         VK_FORMAT_R32G32_SFLOAT; // format data will take (also helps define size
00054                                // of data)
00055     attribute_descriptions[3].offset =
00056         offsetof(Vertex, texture_coords); // where this attribute is defined in
00057                                // the data for a single vertex
00058
00059     return attribute_descriptions;
00060 }
```

Referenced by `PostStage::createGraphicsPipeline()`, and `Rasterizer::createGraphicsPipeline()`.

Here is the caller graph for this function:



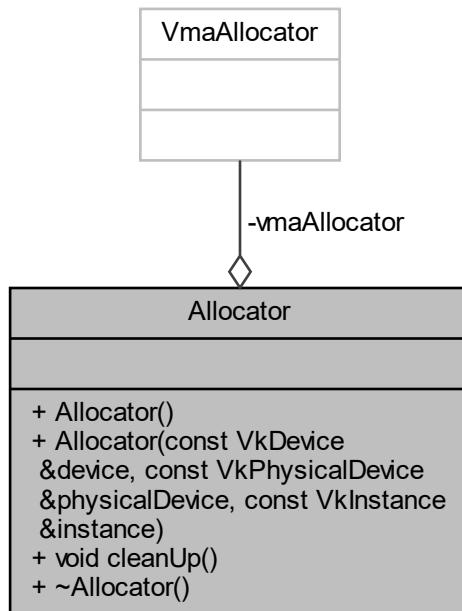
# Chapter 5

## Data Structure Documentation

### 5.1 Allocator Class Reference

```
#include <Allocator.hpp>
```

Collaboration diagram for Allocator:



### Public Member Functions

- `Allocator ()`
- `Allocator (const VkDevice &device, const VkPhysicalDevice &physicalDevice, const VkInstance &instance)`
- `void cleanUp ()`
- `~Allocator ()`

## Private Attributes

- VmaAllocator [vmaAllocator](#)

### 5.1.1 Detailed Description

Definition at line 7 of file [Allocator.hpp](#).

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 Allocator() [1/2]

```
Allocator::Allocator ( )
```

Definition at line 5 of file [Allocator.cpp](#).  
00005 { }

#### 5.1.2.2 Allocator() [2/2]

```
Allocator::Allocator (
    const VkDevice & device,
    const VkPhysicalDevice & physicalDevice,
    const VkInstance & instance )
```

Definition at line 7 of file [Allocator.cpp](#).

```
00009                                     {
00010     // see here:
00011     // https://gpuopen-librariesandsdks.github.io/VulkanMemoryAllocator/html/quick\_start.html
00012     VmaAllocatorCreateInfo allocatorCreateInfo = {};
00013     allocatorCreateInfo.flags = VMA_ALLOCATOR_CREATE_BUFFER_DEVICE_ADDRESS_BIT;
00014     allocatorCreateInfo.vulkanApiVersion = VK_API_VERSION_1_3;
00015     allocatorCreateInfo.physicalDevice = physicalDevice;
00016     allocatorCreateInfo.device = device;
00017     allocatorCreateInfo.instance = instance;
00018
00019     ASSERT_VULKAN(vmaCreateAllocator(&allocatorCreateInfo, &vmaAllocator),
00020                  "Failed to create vma allocator!")
00021 }
```

References [vmaAllocator](#).

#### 5.1.2.3 ~Allocator()

```
Allocator::~Allocator ( )
```

Definition at line 25 of file [Allocator.cpp](#).  
00025 { }

### 5.1.3 Member Function Documentation

#### 5.1.3.1 cleanUp()

```
void Allocator::cleanUp( )
```

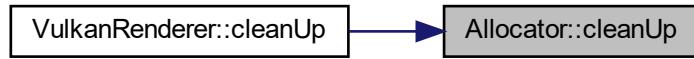
Definition at line 23 of file [Allocator.cpp](#).

```
00023 { vmaDestroyAllocator(vmaAllocator); }
```

References [vmaAllocator](#).

Referenced by [VulkanRenderer::cleanUp\(\)](#).

Here is the caller graph for this function:



### 5.1.4 Field Documentation

#### 5.1.4.1 vmaAllocator

```
VmaAllocator Allocator::vmaAllocator [private]
```

Definition at line 18 of file [Allocator.hpp](#).

Referenced by [Allocator\(\)](#), and [cleanUp\(\)](#).

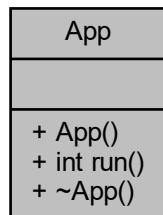
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/memory/[Allocator.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/memory/[Allocator.cpp](#)

## 5.2 App Class Reference

```
#include <App.hpp>
```

Collaboration diagram for App:



### Public Member Functions

- [App \(\)](#)
- [int run \(\)](#)
- [~App \(\)](#)

#### 5.2.1 Detailed Description

Definition at line [2](#) of file [App.hpp](#).

#### 5.2.2 Constructor & Destructor Documentation

##### 5.2.2.1 App()

```
App::App ( )
```

Definition at line [22](#) of file [App.cpp](#).  
00022 { }

##### 5.2.2.2 ~App()

```
App::~App ( )
```

Definition at line [73](#) of file [App.cpp](#).  
00073 { }

## 5.2.3 Member Function Documentation

### 5.2.3.1 run()

```
int App::run ( )
```

Definition at line 24 of file [App.cpp](#).

```
00024     {
00025     int window_width = 1200;
00026     int window_height = 768;
00027
00028     float delta_time = 0.0f;
00029     float last_time = 0.0f;
00030
00031     std::unique_ptr<Window> window =
00032         std::make_unique<Window>(window_width, window_height);
00033     std::unique_ptr<Scene> scene = std::make_unique<Scene>();
00034     std::unique_ptr<GUI> gui = std::make_unique<GUI>(window.get());
00035     std::unique_ptr<Camera> camera = std::make_unique<Camera>();
00036
00037     VulkanRenderer vulkan_renderer{window.get(), scene.get(), gui.get(),
00038                                     camera.get()};
00039
00040     while (!window->get_should_close()) {
00041         // poll all events incoming from user
00042         glfwPollEvents();
00043
00044         // handle events for the camera
00045         camera->key_control(window->get_keys(), delta_time);
00046         camera->mouse_control(window->get_x_change(), window->get_y_change());
00047
00048         float now = static_cast<float>(glfwGetTime());
00049         delta_time = now - last_time;
00050         last_time = now;
00051
00052         scene->update_user_input(gui.get());
00053
00054         vulkan_renderer.updateStateDueToUserInput(gui.get());
00055         vulkan_renderer.updateUniforms(scene.get(), camera.get(), window.get());
00056
00057         //// retrieve updates from the UI
00058         gui->render();
00059
00060         vulkan_renderer.drawFrame();
00061     }
00062
00063     vulkan_renderer.finishAllRenderCommands();
00064
00065     scene->cleanUp();
00066     gui->cleanUp();
00067     window->cleanUp();
00068     vulkan_renderer.cleanUp();
00069
00070     return EXIT_SUCCESS;
00071 }
```

Referenced by [main\(\)](#).

Here is the caller graph for this function:



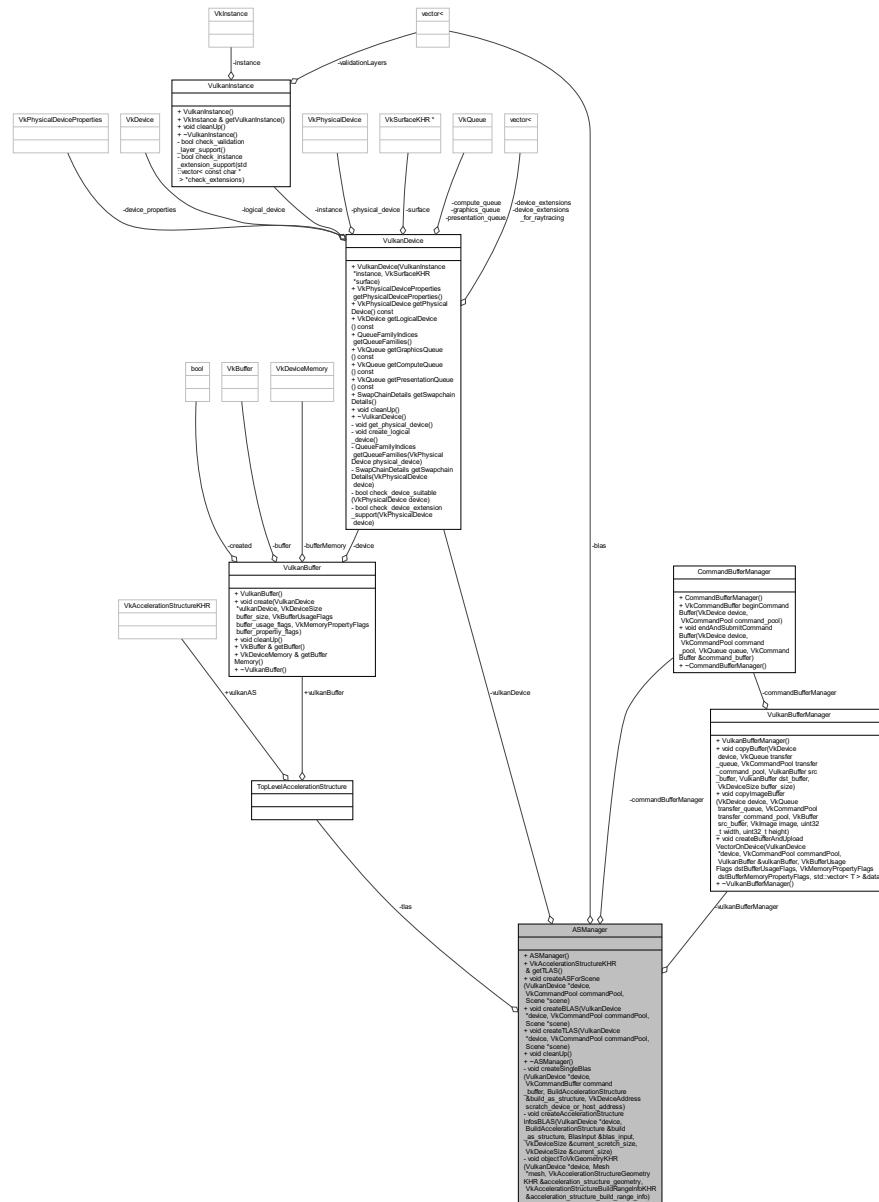
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/app/[App.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/app/[App.cpp](#)

## 5.3 ASManager Class Reference

```
#include <ASManager.hpp>
```

## Collaboration diagram for ASManager:



## Public Member Functions

- `ASManager ()`
  - `VkAccelerationStructureKHR & getTLAS ()`
  - `void createASForScene (VulkanDevice *device, VkCommandPool commandPool, Scene *scene)`
  - `void createBLAS (VulkanDevice *device, VkCommandPool commandPool, Scene *scene)`
  - `void createTLAS (VulkanDevice *device, VkCommandPool commandPool, Scene *scene)`
  - `void cleanUp ()`
  - `~ASManager ()`

## Private Member Functions

- void `createSingleBlas` (`VulkanDevice` \*device, `VkCommandBuffer` command\_buffer, `BuildAccelerationStructure` &build\_as\_structure, `VkDeviceAddress` scratch\_device\_or\_host\_address)
- void `createAccelerationStructureInfosBLAS` (`VulkanDevice` \*device, `BuildAccelerationStructure` &build\_as\_structure, `BlasInput` &blas\_input, `VkDeviceSize` &current\_scratch\_size, `VkDeviceSize` &current\_size)
- void `objectToVkGeometryKHR` (`VulkanDevice` \*device, `Mesh` \*mesh, `VkAccelerationStructureGeometryKHR` &acceleration\_structure\_geometry, `VkAccelerationStructureBuildRangeInfoKHR` &acceleration\_structure\_build\_range\_info)

## Private Attributes

- `VulkanDevice` \* `vulkanDevice` {VK\_NULL\_HANDLE}
- `CommandBufferManager` `commandBufferManager`
- `VulkanBufferManager` `vulkanBufferManager`
- `std::vector<BottomLevelAccelerationStructure>` `blas`
- `TopLevelAccelerationStructure` `tlas`

### 5.3.1 Detailed Description

Definition at line 22 of file `ASManager.hpp`.

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 ASManager()

```
ASManager::ASManager ( )
```

Definition at line 3 of file `ASManager.cpp`.  
00003 { }

#### 5.3.2.2 ~ASManager()

```
ASManager::~ASManager ( )
```

Definition at line 375 of file `ASManager.cpp`.  
00375 { }

### 5.3.3 Member Function Documentation

### 5.3.3.1 cleanUp()

```
void ASManager::cleanUp ( )
```

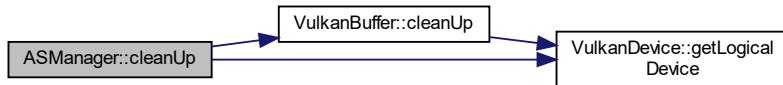
Definition at line 356 of file [ASManager.cpp](#).

```
00356     {
00357     PFN_vkDestroyAccelerationStructureKHR pvkDestroyAccelerationStructureKHR =
00358         (PFN_vkDestroyAccelerationStructureKHR)vkGetDeviceProcAddr(
00359             vulkanDevice->getLogicalDevice(),
00360             "vkDestroyAccelerationStructureKHR");
00361
00362     pvkDestroyAccelerationStructureKHR(vulkanDevice->getLogicalDevice(),
00363                                         tlas.vulkanAS, nullptr);
00364
00365     tlas.vulkanBuffer.cleanUp();
00366
00367     for (size_t index = 0; index < blas.size(); index++) {
00368         pvkDestroyAccelerationStructureKHR(vulkanDevice->getLogicalDevice(),
00369             blas[index].vulkanAS, nullptr);
00370
00371         blas[index].vulkanBuffer.cleanUp();
00372     }
00373 }
```

References [blas](#), [VulkanBuffer::cleanUp\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [tlas](#), [TopLevelAccelerationStructure::vulkanAS](#), [TopLevelAccelerationStructure::vulkanBuffer](#), and [vulkanDevice](#).

Referenced by [VulkanRenderer::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.3.3.2 createAccelerationStructureInfosBLAS()

```
void ASManager::createAccelerationStructureInfosBLAS (
    VulkanDevice * device,
    BuildAccelerationStructure & build_as_structure,
    BlasInput & blas_input,
    VkDeviceSize & current_scratch_size,
    VkDeviceSize & current_size ) [private]
```

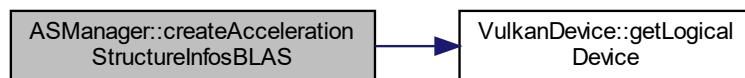
Definition at line 420 of file [ASManager.cpp](#).

```
00423     {
00424     PFN_vkGetAccelerationStructureBuildSizesKHR
00425         pvkGetAccelerationStructureBuildSizesKHR =
00426             (PFN_vkGetAccelerationStructureBuildSizesKHR)vkGetDeviceProcAddr(
00427                 device->getLogicalDevice\(\),
00428                 "vkGetAccelerationStructureBuildSizesKHR");
00429
00430     build_as_structure.build\_info.sType =
00431         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
00432     build_as_structure.build\_info.type =
00433         VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
00434     build_as_structure.build\_info.flags =
00435         VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
00436     build_as_structure.build\_info.mode =
00437         VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
00438     build_as_structure.build\_info.geometryCount =
00439         static_cast<uint32_t>(blas_input.as\_geometry.size());
00440     build_as_structure.build\_info.pGeometries = blas_input.as\_geometry.data();
00441
00442     build_as_structure.range\_info = blas_input.as\_build\_offset\_info.data();
00443
00444     build_as_structure.size\_info.sType =
00445         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR;
00446
00447     std::vector<uint32_t> max_primitive_cnt(
00448         blas_input.as\_build\_offset\_info.size());
00449
00450     for (uint32_t temp = 0;
00451         temp < static_cast<uint32_t>(blas_input.as\_build\_offset\_info.size());
00452         temp++)
00453     max_primitive_cnt[temp] =
00454         blas_input.as\_build\_offset\_info[temp].primitiveCount;
00455
00456     pvkGetAccelerationStructureBuildSizesKHR(
00457         device->getLogicalDevice\(\),
00458         VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR,
00459         &build_as_structure.build\_info, max_primitive_cnt.data(),
00460         &build_as_structure.size\_info);
00461
00462     current_size = build_as_structure.size\_info.accelerationStructureSize;
00463     current_scratch_size = build_as_structure.size\_info.buildScratchSize;
00464 }
```

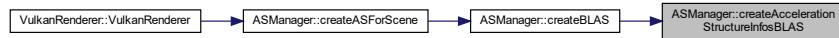
References [BlasInput::as\\_build\\_offset\\_info](#), [BlasInput::as\\_geometry](#), [BuildAccelerationStructure::build\\_info](#), [VulkanDevice::getLogicalDevice\(\)](#), [BuildAccelerationStructure::range\\_info](#), and [BuildAccelerationStructure::size\\_info](#).

Referenced by [createBLAS\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.3.3.3 createASForScene()

```
void ASManager::createASForScene (
    VulkanDevice * device,
    VkCommandPool commandPool,
    Scene * scene )
```

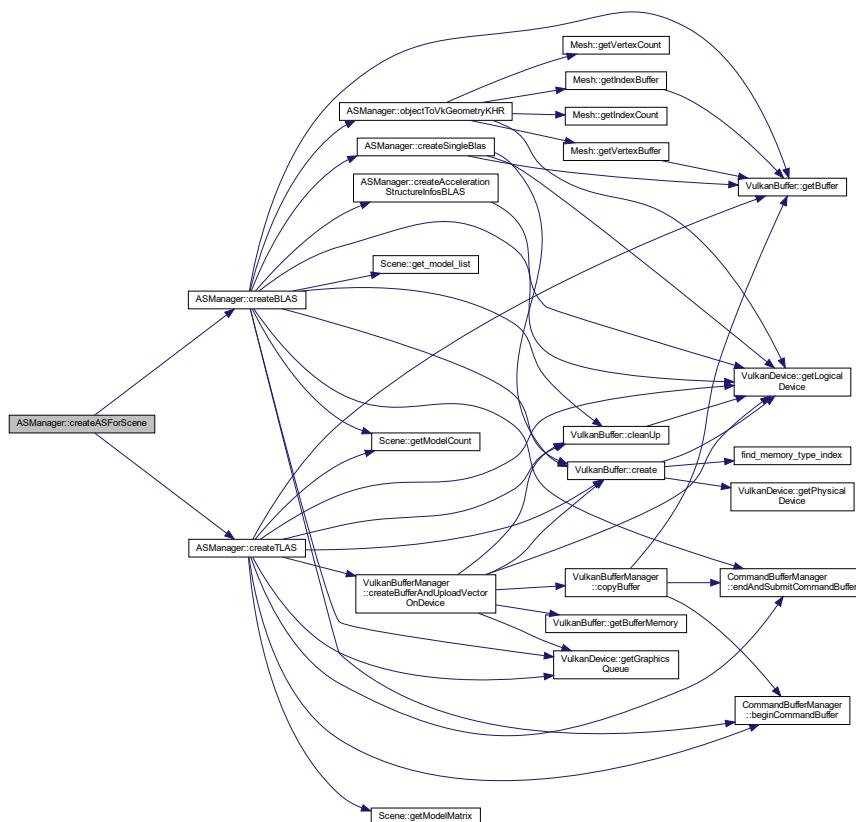
Definition at line 5 of file [ASManager.cpp](#).

```
00006
00007     this->vulkanDevice = device;
00008     createBLAS(device, commandPool, scene);
00009     createTLAS(device, commandPool, scene);
00010 }
```

References [createBLAS\(\)](#), [createTLAS\(\)](#), and [vulkanDevice](#).

Referenced by [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.3.3.4 createBLAS()

```
void ASManager::createBLAS (
    VulkanDevice * device,
    VkCommandPool commandPool,
    Scene * scene )
```

Definition at line 12 of file [ASManager.cpp](#).

```

00013     {
00014     // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
00015     // all functionality from extensions has to be loaded in the beginning
00016     // we need a reference to the device location of our geometry laying on the
00017     // graphics card we already uploaded objects and created vertex and index
00018     // buffers respectively
00019
00020     PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
00021         (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
00022             device->getLogicalDevice\(\), "vkGetBufferDeviceAddress");
00023
00024     std::vector<BlasInput> blas_input(scene->getModelCount\(\));
00025
00026     for (uint32_t model_index = 0;
00027         model_index < static_cast<uint32_t>(scene->getModelCount\(\));
00028         model_index++) {
00029         std::shared_ptr<Model> mesh_model = scene->get\_model\_list\(\)[model_index];
00030         // blas\_input.emplace\_back\(\);
00031         blas_input[model_index].as_geometry.reserve(mesh_model->getMeshCount());
00032         blas_input[model_index].as_build_offset_info.reserve(
00033             mesh_model->getMeshCount());
00034
00035         for (size_t mesh_index = 0; mesh_index < mesh_model->getMeshCount();
00036             mesh_index++) {
00037             VkAccelerationStructureGeometryKHR acceleration_structure_geometry{};
00038             VkAccelerationStructureBuildRangeInfoKHR
00039                 acceleration_structure_build_range_info{};
00040
00041             objectToVkGeometryKHR(device, mesh_model->getMesh(mesh_index),
00042                     acceleration_structure_geometry,
00043                     acceleration_structure_build_range_info);
00044             // this only specifies the acceleration structure
00045             // we are building it in the end for the whole model with the build
00046             // command
00047
00048             blas_input[model_index].as_geometry.push_back(
00049                 acceleration_structure_geometry);
00050             blas_input[model_index].as_build_offset_info.push_back(
00051                 acceleration_structure_build_range_info);
00052         }
00053     }
00054
00055     std::vector<BuildAccelerationStructure> build_as_structures;
00056     build_as_structures.resize(scene->getModelCount\(\));
00057
00058     VkDeviceSize max_scratch_size = 0;
00059     VkDeviceSize total_size_all_BLAS = 0;
00060
00061     for (unsigned int i = 0; i < scene->getModelCount\(\); i++) {
00062         VkDeviceSize current_scratch_size = 0;
00063         VkDeviceSize current_size = 0;
00064
```

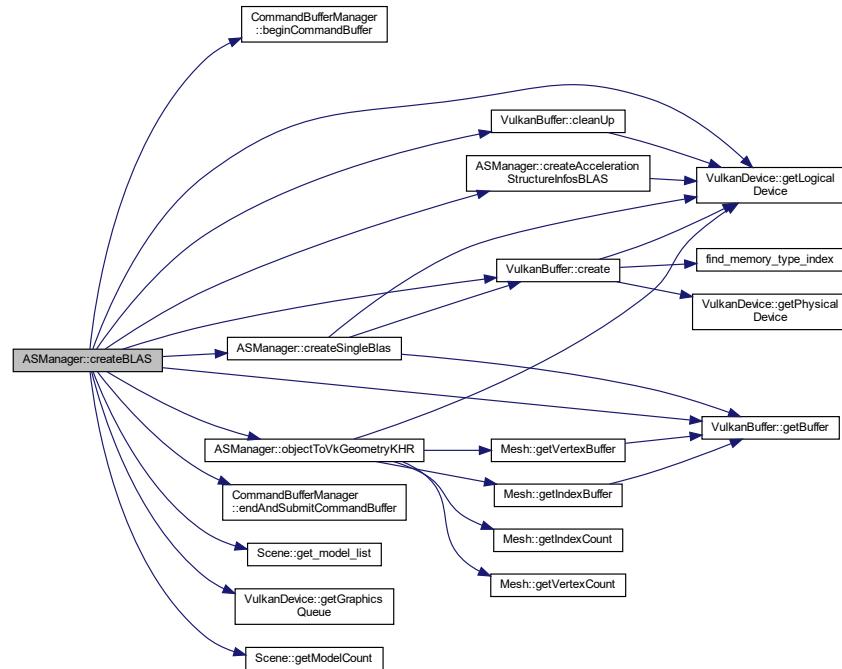
```

00065     createAccelerationStructureInfosBLAS(device, build_as_structures[i],
00066                                         blas_input[i], current_scratch_size,
00067                                         current_size);
00068
00069     total_size_all_BLAS += current_size;
00070     max_scratch_size = std::max(max_scratch_size, current_scratch_size);
00071 }
00072
00073 VulkanBuffer scratchBuffer;
00074
00075 scratchBuffer.create(device, max_scratch_size,
00076                       VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
00077                           VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00078                           VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00079                           VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
00080                           VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00081
00082 VkBufferDeviceAddressInfo scratch_buffer_device_address_info{};
00083 scratch_buffer_device_address_info.sType =
00084     VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00085 scratch_buffer_device_address_info.buffer = scratchBuffer.getBuffer();
00086
00087 VkDeviceAddress scratch_buffer_address = pvkGetBufferDeviceAddressKHR(
00088     device->getLogicalDevice(), &scratch_buffer_device_address_info);
00089
00090 VkDeviceOrHostAddressKHR scratch_device_or_host_address{};
00091 scratch_device_or_host_address.deviceAddress = scratch_buffer_address;
00092
00093 VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
00094     device->getLogicalDevice(), commandPool);
00095
00096 for (size_t i = 0; i < scene->getModelCount(); i++) {
00097     createSingleBlas(device, command_buffer, build_as_structures[i],
00098                      scratch_buffer_address);
00099
00100     VkMemoryBarrier barrier;
00101     barrier.pNext = nullptr;
00102     barrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
00103     barrier.srcAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
00104     barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR;
00105
00106     vkCmdPipelineBarrier(command_buffer,
00107                           VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
00108                           VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
00109                           0, 1, &barrier, 0, nullptr, 0, nullptr);
00110 }
00111
00112 commandBufferManager.endAndSubmitCommandBuffer(
00113     device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
00114     command_buffer);
00115
00116 for (auto& b : build_as_structures) {
00117     blas.emplace_back(b.single_blas);
00118 }
00119
00120 scratchBuffer.cleanUp();
00121 }
```

References [CommandBufferManager::beginCommandBuffer\(\)](#), [blas](#), [VulkanBuffer::cleanUp\(\)](#), [commandBufferManager](#), [VulkanBuffer::create\(\)](#), [createAccelerationStructureInfosBLAS\(\)](#), [createSingleBlas\(\)](#), [CommandBufferManager::endAndSubmitCommandBuffer\(\)](#), [Scene::get\\_model\\_list\(\)](#), [VulkanBuffer::getBuffer\(\)](#), [VulkanDevice::getGraphicsQueue\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [Scene::getModelCount\(\)](#), and [objectToVkGeometryKHR\(\)](#).

Referenced by [createASForScene\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.3.3.5 createSingleBlas()

```
void ASManager::createSingleBlas (
    VulkanDevice * device,
    VkCommandBuffer command_buffer,
    BuildAccelerationStructure & build_as_structure,
    VkDeviceAddress scratch_device_or_host_address ) [private]
```

Definition at line 377 of file [ASManager.cpp](#).

```
00380                                     {  
00381     PFN_vkCreateAccelerationStructureKHR pvkCreateAccelerationStructureKHR =  
00382         (PFN_vkCreateAccelerationStructureKHR)vkGetDeviceProcAddr(  
00383             device->getLogicalDevice(), "vkCreateAccelerationStructureKHR");  
00384  
00385     PFN_vkCmdBuildAccelerationStructuresKHR pvkCmdBuildAccelerationStructuresKHR =  
00386         (PFN_vkCmdBuildAccelerationStructuresKHR)vkGetDeviceProcAddr(  
00387             device->getLogicalDevice(), "vkCmdBuildAccelerationStructuresKHR");  
00388  
00389     VkAccelerationStructureCreateInfoKHR acceleration_structure_create_info{};  
00390     acceleration_structure_create_info.sType =
```

```

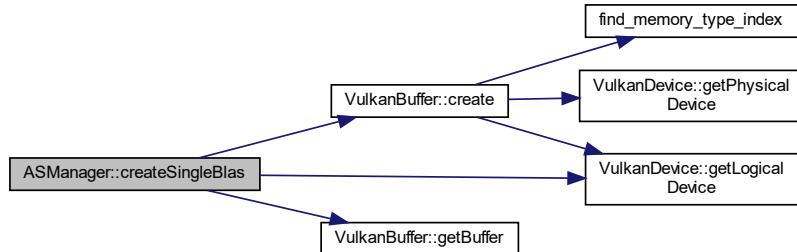
00391     VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
00392     acceleration_structure_create_info.type =
00393         VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
00394     acceleration_structure_create_info.size =
00395         build_as_structure.size_info.accelerationStructureSize;
00396     VulkanBuffer* blasVulkanBuffer = build_as_structure.single_blas.vulkanBuffer;
00397     blasVulkanBuffer.create(
00398         device, build_as_structure.size_info.accelerationStructureSize,
00399         VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR |
00400             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00401             VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00402             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
00403             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00404
00405     acceleration_structure_create_info.buffer = blasVulkanBuffer.getBuffer();
00406     VkAccelerationStructureKHR& blas_as = build_as_structure.single_blas.vulkanAS;
00407     pvkCreateAccelerationStructureKHR(device->getLogicalDevice(),
00408                                         &acceleration_structure_create_info,
00409                                         nullptr, &blas_as);
00410
00411     build_as_structure.build_info.dstAccelerationStructure = blas_as;
00412     build_as_structure.build_info.scratchData.deviceAddress =
00413         scratch_device_or_host_address;
00414
00415     pvkCmdBuildAccelerationStructuresKHR(command_buffer, 1,
00416                                         &build_as_structure.build_info,
00417                                         &build_as_structure.range_info);
00418 }

```

References [BuildAccelerationStructure::build\\_info](#), [VulkanBuffer::create\(\)](#), [VulkanBuffer::getBuffer\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [BuildAccelerationStructure::range\\_info](#), [BuildAccelerationStructure::single\\_blas](#), [BuildAccelerationStructure::size\\_info](#), [BottomLevelAccelerationStructure::vulkanAS](#), and [BottomLevelAccelerationStructure::vulkanBuffer](#).

Referenced by [createBLAS\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.3.3.6 createTLAS()

```
void ASManager::createTLAS (
    VulkanDevice * device,
    VkCommandPool commandPool,
    Scene * scene )
```

Definition at line 123 of file [ASManager.cpp](#).

```
00124     {
00125         // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
00126         // all functionality from extensions has to be loaded in the beginning
00127         // we need a reference to the device location of our geometry laying on the
00128         // graphics card we already uploaded objects and created vertex and index
00129         // buffers respectively
00130         PFN_vkGetAccelerationStructureBuildSizesKHR =
00131             pvkGetAccelerationStructureBuildSizesKHR =
00132                 (PFN_vkGetAccelerationStructureBuildSizesKHR)vkGetDeviceProcAddr(
00133                     device->getLogicalDevice(),
00134                     "vkGetAccelerationStructureBuildSizesKHR");
00135
00136         PFN_vkCreateAccelerationStructureKHR pvkCreateAccelerationStructureKHR =
00137             (PFN_vkCreateAccelerationStructureKHR)vkGetDeviceProcAddr(
00138                 device->getLogicalDevice(), "vkCreateAccelerationStructureKHR");
00139
00140         PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
00141             (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
00142                 device->getLogicalDevice(), "vkGetBufferDeviceAddress");
00143
00144         PFN_vkCmdBuildAccelerationStructuresKHR pvkCmdBuildAccelerationStructuresKHR =
00145             (PFN_vkCmdBuildAccelerationStructuresKHR)vkGetDeviceProcAddr(
00146                 device->getLogicalDevice(), "vkCmdBuildAccelerationStructuresKHR");
00147
00148         PFN_vkGetAccelerationStructureDeviceAddressKHR
00149             pvkGetAccelerationStructureDeviceAddressKHR =
00150                 (PFN_vkGetAccelerationStructureDeviceAddressKHR)vkGetDeviceProcAddr(
00151                     device->getLogicalDevice(),
00152                     "vkGetAccelerationStructureDeviceAddressKHR");
00153
00154         std::vector<VkAccelerationStructureInstanceKHR> tlas_instances;
00155         tlas_instances.reserve(scene->getModelCount());
00156
00157     for (size_t model_index = 0; model_index < scene->getModelCount();
00158         model_index++) {
00159         // glm uses column major matrices so transpose it for Vulkan want row major
00160         // here
00161         glm::mat4 transpose_transform =
00162             glm::transpose(scene->getModelMatrix(static_cast<int>(model_index)));
00163         VkTransformMatrixKHR out_matrix;
00164         memcpy(&out_matrix, &transpose_transform, sizeof(VkTransformMatrixKHR));
00165
00166         VkAccelerationStructureDeviceAddressInfoKHR
00167             acceleration_structure_device_address_info{};
00168         acceleration_structure_device_address_info.sType =
00169             VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_DEVICE_ADDRESS_INFO_KHR;
00170         acceleration_structure_device_address_info.accelerationStructure =
00171             blas[model_index].vulkanAS;
00172
00173         VkDeviceAddress acceleration_structure_device_address =
00174             pvkGetAccelerationStructureDeviceAddressKHR(
00175                 device->getLogicalDevice(),
00176                 &acceleration_structure_device_address_info);
00177
00178         VkAccelerationStructureInstanceKHR geometry_instance{};
00179         geometry_instance.transform = out_matrix;
00180         geometry_instance.instanceCustomIndex =
00181             model_index; // gl_InstanceCustomIndexEXT
00182         geometry_instance.mask = 0xFF;
00183         geometry_instance.instanceShaderBindingTableRecordOffset = 0;
00184         geometry_instance.flags =
00185             VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
00186         geometry_instance.accelerationStructureReference =
00187             acceleration_structure_device_address;
00188         geometry_instance.instanceShaderBindingTableRecordOffset =
00189             0; // same hit group for all objects
00190
00191         tlas_instances.emplace_back(geometry_instance);
00192     }
00193
00194     VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
00195         device->getLogicalDevice(), commandPool);
00196
00197     VulkanBuffer geometryInstanceBuffer;
```

```

00199     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00200         device, commandPool, geometryInstanceBuffer,
00201         VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00202             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00203             VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00204         VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
00205             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
00206         tlas_instances);
00207
00208     VkBufferDeviceAddressInfo geometry_instance_buffer_device_address_info{};
00209     geometry_instance_buffer_device_address_info.sType =
00210         VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00211     geometry_instance_buffer_device_address_info.buffer =
00212         geometryInstanceBuffer.getBuffer();
00213
00214     VkDeviceAddress geometry_instance_buffer_address =
00215         pvkGetBufferDeviceAddressKHR(
00216             device->getLogicalDevice(),
00217             &geometry_instance_buffer_device_address_info);
00218
00219 // Make sure the copy of the instance buffer are copied before triggering the
00220 // acceleration structure build
00221     VkMemoryBarrier barrier;
00222     barrier.pNext = nullptr;
00223     barrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
00224     barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00225     barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
00226     vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00227             VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
00228             0, 1, &barrier, 0, nullptr, 0, nullptr);
00229
00230     VkAccelerationStructureGeometryInstancesDataKHR
00231         acceleration_structure_geometry_instances_data{};
00232     acceleration_structure_geometry_instances_data.sType =
00233         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR;
00234     acceleration_structure_geometry_instances_data.pNext = nullptr;
00235     acceleration_structure_geometry_instances_data.data.deviceAddress =
00236         geometry_instance_buffer_address;
00237
00238     VkAccelerationStructureGeometryKHR topAS_acceleration_structure_geometry{};
00239     topAS_acceleration_structure_geometry.sType =
00240         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
00241     topAS_acceleration_structure_geometry.pNext = nullptr;
00242     topAS_acceleration_structure_geometry.geometryType =
00243         VK_GEOMETRY_TYPE_INSTANCES_KHR;
00244     topAS_acceleration_structure_geometry.geometry.instances =
00245         acceleration_structure_geometry_instances_data;
00246
00247 // find sizes
00248     VkAccelerationStructureBuildGeometryInfoKHR
00249         acceleration_structure_build_geometry_info{};
00250     acceleration_structure_build_geometry_info.sType =
00251         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
00252     acceleration_structure_build_geometry_info.pNext = nullptr;
00253     acceleration_structure_build_geometry_info.type =
00254         VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
00255     acceleration_structure_build_geometry_info.flags =
00256         VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
00257     acceleration_structure_build_geometry_info.mode =
00258         VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
00259     acceleration_structure_build_geometry_info.srcAccelerationStructure =
00260         VK_NULL_HANDLE;
00261     acceleration_structure_build_geometry_info.geometryCount = 1;
00262     acceleration_structure_build_geometry_info.pGeometries =
00263         &topAS_acceleration_structure_geometry;
00264
00265     VkAccelerationStructureBuildSizesInfoKHR
00266         acceleration_structure_build_sizes_info{};
00267     acceleration_structure_build_sizes_info.sType =
00268         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR;
00269     acceleration_structure_build_sizes_info.pNext = nullptr;
00270     acceleration_structure_build_sizes_info.accelerationStructureSize = 0;
00271     acceleration_structure_build_sizes_info.updateScratchSize = 0;
00272     acceleration_structure_build_sizes_info.buildScratchSize = 0;
00273
00274     uint32_t count_instance = static_cast<uint32_t>(tlas_instances.size());
00275     pvkGetAccelerationStructureBuildSizesKHR(
00276         device->getLogicalDevice(), VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_KHR,
00277         &acceleration_structure_build_geometry_info, &count_instance,
00278         &acceleration_structure_build_sizes_info);
00279
00280 // now we got the sizes
00281     VulkanBuffer& tlasVulkanBuffer = tlas.vulkanBuffer;
00282     tlasVulkanBuffer.create(
00283         device, acceleration_structure_build_sizes_info.accelerationStructureSize,
00284         VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR |
00285             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |

```

```

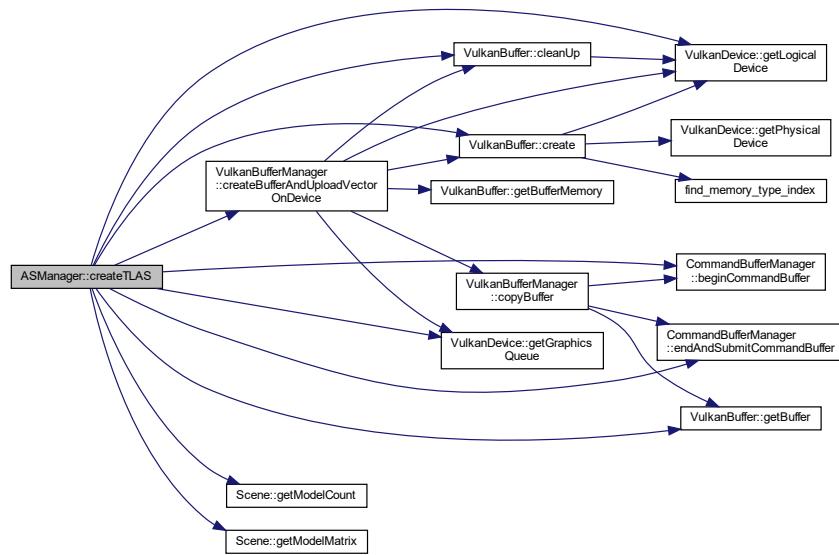
00286         VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00287         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00288         VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT);
00289
00290     VkAccelerationStructureCreateInfoKHR acceleration_structure_create_info{};
00291     acceleration_structure_create_info.sType =
00292         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
00293     acceleration_structure_create_info.pNext = nullptr;
00294     acceleration_structure_create_info.createFlags = 0;
00295     acceleration_structure_create_info.buffer = tlasVulkanBuffer.getBuffer();
00296     acceleration_structure_create_info.offset = 0;
00297     acceleration_structure_create_info.size =
00298         acceleration_structure_build_sizes_info.accelerationStructureSize;
00299     acceleration_structure_create_info.type =
00300         VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
00301     acceleration_structure_create_info.deviceAddress = 0;
00302
00303     VkAccelerationStructureKHR& tLAS = tlas.vulkanAS;
00304     pvkCreateAccelerationStructureKHR(device->getLogicalDevice(),
00305                                         &acceleration_structure_create_info,
00306                                         nullptr, &tLAS);
00307
00308     VulkanBuffer scratchBuffer;
00309
00310     scratchBuffer.create(device,
00311                           acceleration_structure_build_sizes_info.buildScratchSize,
00312                           VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
00313                           VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00314                           VK_BUFFER_USAGE_TRANSFER_DST_BIT |
00315                           VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00316                           VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT);
00317
00318     VkBufferDeviceAddressInfo scratch_buffer_device_address_info{};
00319     scratch_buffer_device_address_info.sType =
00320         VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00321     scratch_buffer_device_address_info.buffer = scratchBuffer.getBuffer();
00322
00323     VkDeviceAddress scratch_buffer_address = pvkGetBufferDeviceAddressKHR(
00324         device->getLogicalDevice(), &scratch_buffer_device_address_info);
00325
00326 // update build info
00327     acceleration_structure_build_geometry_info.scratchData.deviceAddress =
00328         scratch_buffer_address;
00329     acceleration_structure_build_geometry_info.srcAccelerationStructure =
00330         VK_NULL_HANDLE;
00331     acceleration_structure_build_geometry_info.dstAccelerationStructure = tLAS;
00332
00333     VkAccelerationStructureBuildRangeInfoKHR
00334         acceleration_structure_build_range_info{};
00335     acceleration_structure_build_range_info.primitiveCount =
00336         scene->getModelCount();
00337     acceleration_structure_build_range_info.primitiveOffset = 0;
00338     acceleration_structure_build_range_info.firstVertex = 0;
00339     acceleration_structure_build_range_info.transformOffset = 0;
00340
00341     VkAccelerationStructureBuildRangeInfoKHR*
00342         acceleration_structure_build_range_infos =
00343             &acceleration_structure_build_range_info;
00344
00345     pvkCmdBuildAccelerationStructuresKHR(
00346         command_buf, 1, &acceleration_structure_build_geometry_info,
00347         &acceleration_structure_build_range_infos);
00348
00349     commandBufferManager.endAndSubmitCommandBuffer(
00350         device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
00351         command_buf);
00352     scratchBuffer.cleanUp();
00353     geometryInstanceBuffer.cleanUp();
00354 }

```

References [CommandBufferManager::beginCommandBuffer\(\)](#), [blas](#), [VulkanBuffer::cleanUp\(\)](#), [commandBufferManager](#), [VulkanBuffer::create\(\)](#), [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [CommandBufferManager::endAndSubmitCommandBuffer\(\)](#), [VulkanBuffer::getBuffer\(\)](#), [VulkanDevice::getGraphicsQueue\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [Scene::getModelCount\(\)](#), [Scene::getModelMatrix\(\)](#), [tlas](#), [TopLevelAccelerationStructure::vulkanAS](#), [TopLevelAccelerationStructure::vulkanBuffer](#), and [vulkanBufferManager](#).

Referenced by [createASForScene\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.3.3.7 getTLAS()

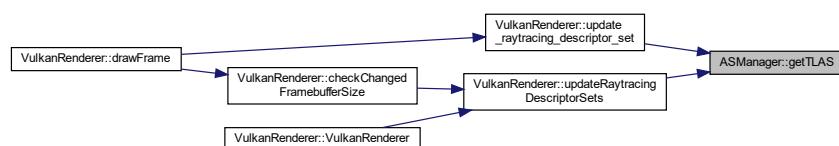
```
VkAccelerationStructureKHR & ASManager::getTLAS ( ) [inline]
```

Definition at line 26 of file [ASManager.hpp](#).  
00026 { return tlas.vulkanAS; };

References [tlas](#), and [TopLevelAccelerationStructure::vulkanAS](#).

Referenced by [VulkanRenderer::update\\_raytracing\\_descriptor\\_set\(\)](#), and [VulkanRenderer::updateRaytracingDescriptorSets\(\)](#).

Here is the caller graph for this function:



### 5.3.3.8 objectToVkGeometryKHR()

```
void ASManager::objectToVkGeometryKHR (
    VulkanDevice * device,
    Mesh * mesh,
    VkAccelerationStructureGeometryKHR & acceleration_structure_geometry,
    VkAccelerationStructureBuildRangeInfoKHR & acceleration_structure_build_range_<-
info ) [private]
```

Definition at line 466 of file [ASManager.cpp](#).

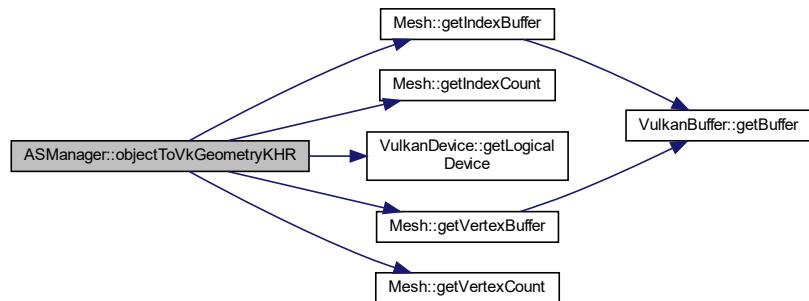
```
00470 {
00471     // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
00472     // all functionality from extensions has to be loaded in the beginning
00473     // we need a reference to the device location of our geometry laying on the
00474     // graphics card we already uploaded objects and created vertex and index
00475     // buffers respectively
00476     PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
00477         (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
00478             device->getLogicalDevice(), "vkGetBufferDeviceAddress");
00479
00480     // all starts with the address of our vertex and index data we already
00481     // uploaded in buffers earlier when loading the meshes/models
00482     VkBufferDeviceAddressInfo vertex_buffer_device_address_info{};
00483     vertex_buffer_device_address_info.sType =
00484         VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00485     vertex_buffer_device_address_info.buffer = mesh->getVertexBuffer();
00486     vertex_buffer_device_address_info.pNext = nullptr;
00487
00488     VkBufferDeviceAddressInfo index_buffer_device_address_info{};
00489     index_buffer_device_address_info.sType =
00490         VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00491     index_buffer_device_address_info.buffer = mesh->getIndexBuffer();
00492     index_buffer_device_address_info.pNext = nullptr;
00493
00494     // receiving address to move on
00495     VkDeviceAddress vertex_buffer_address = pvkGetBufferDeviceAddressKHR(
00496         device->getLogicalDevice(), &vertex_buffer_device_address_info);
00497     VkDeviceAddress index_buffer_address = pvkGetBufferDeviceAddressKHR(
00498         device->getLogicalDevice(), &index_buffer_device_address_info);
00499
00500     // convert to const address for further processing
00501     VkDeviceOrHostAddressConstKHR vertex_device_or_host_address_const{};
00502     vertex_device_or_host_address_const.deviceAddress = vertex_buffer_address;
00503
00504     VkDeviceOrHostAddressConstKHR index_device_or_host_address_const{};
00505     index_device_or_host_address_const.deviceAddress = index_buffer_address;
00506
00507     VkAccelerationStructureGeometryTrianglesDataKHR
00508         acceleration_structure_triangles_data{};
00509     acceleration_structure_triangles_data.sType =
00510         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR;
00511     acceleration_structure_triangles_data.pNext = nullptr;
00512     acceleration_structure_triangles_data.vertexFormat =
00513         VK_FORMAT_R32G32B32_SFLOAT;
00514     acceleration_structure_triangles_data.vertexData =
00515         vertex_device_or_host_address_const;
00516     acceleration_structure_triangles_data.vertexStride = sizeof(Vertex);
00517     acceleration_structure_triangles_data.maxVertex = mesh->getVertexCount();
00518     acceleration_structure_triangles_data.indexType = VK_INDEX_TYPE_UINT32;
00519     acceleration_structure_triangles_data.indexData =
00520         index_device_or_host_address_const;
00521
00522     // can also be instances or AABBs; not covered here
00523     // but to identify as triangles put it into these struct
00524     VkAccelerationStructureGeometryDataKHR acceleration_structure_geometry_data{};
00525     acceleration_structure_geometry_data.triangles =
00526         acceleration_structure_triangles_data;
00527
00528     acceleration_structure_geometry.sType =
00529         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
00530     acceleration_structure_geometry.pNext = nullptr;
00531     acceleration_structure_geometry.geometryType = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
00532     acceleration_structure_geometry.geometry =
00533         acceleration_structure_geometry_data;
00534     acceleration_structure_geometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR;
00535
00536     // we have triangles so divide the number of vertices with 3!!
00537     // for our simple case a no brainer
00538     // take entire data to build BLAS
00539     // number of indices is truly the stick point here
00540     acceleration_structure_build_range_info.primitiveCount =
00541         mesh->getIndexCount() / 3;
```

```
00542     acceleration_structure_build_range_info.primitiveOffset = 0;
00543     acceleration_structure_build_range_info.firstVertex = 0;
00544     acceleration_structure_build_range_info.transformOffset = 0;
00545 }
```

References [Mesh::getIndexBuffer\(\)](#), [Mesh::getIndexCount\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [Mesh::getVertexBuffer\(\)](#), and [Mesh::getVertexCount\(\)](#).

Referenced by [createBLAS\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.3.4 Field Documentation

### 5.3.4.1 blas

```
std::vector<BottomLevelAccelerationStructure> ASManager::blas [private]
```

Definition at line 46 of file [ASManager.hpp](#).

Referenced by [cleanUp\(\)](#), [createBLAS\(\)](#), and [createTLAS\(\)](#).

### 5.3.4.2 commandBufferManager

```
CommandBufferManager ASManager::commandBufferManager [private]
```

Definition at line 43 of file [ASManager.hpp](#).

Referenced by [createBLAS\(\)](#), and [createTLAS\(\)](#).

### 5.3.4.3 tlas

`TopLevelAccelerationStructure ASManager::tlas [private]`

Definition at line 47 of file [ASManager.hpp](#).

Referenced by [cleanUp\(\)](#), [createTLAS\(\)](#), and [getTLAS\(\)](#).

### 5.3.4.4 vulkanBufferManager

`VulkanBufferManager ASManager::vulkanBufferManager [private]`

Definition at line 44 of file [ASManager.hpp](#).

Referenced by [createTLAS\(\)](#).

### 5.3.4.5 vulkanDevice

`VulkanDevice* ASManager::vulkanDevice {VK_NULL_HANDLE} [private]`

Definition at line 42 of file [ASManager.hpp](#).

Referenced by [cleanUp\(\)](#), and [createASForScene\(\)](#).

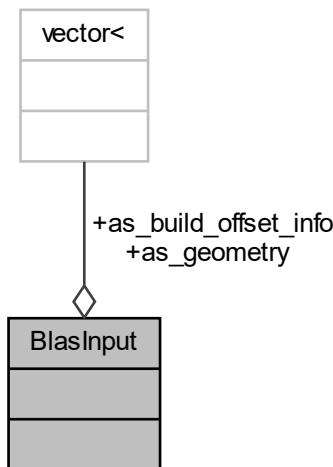
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/[ASManager.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/accelerationStructures/[ASManager.cpp](#)

## 5.4 BlasInput Struct Reference

#include <ASManager.hpp>

Collaboration diagram for BlasInput:



## Data Fields

- std::vector< VkAccelerationStructureGeometryKHR > [as\\_geometry](#)
- std::vector< VkAccelerationStructureBuildRangeInfoKHR > [as\\_build\\_offset\\_info](#)

### 5.4.1 Detailed Description

Definition at line 17 of file [ASManager.hpp](#).

### 5.4.2 Field Documentation

#### 5.4.2.1 [as\\_build\\_offset\\_info](#)

```
std::vector<VkAccelerationStructureBuildRangeInfoKHR> BlasInput::as_build_offset_info
```

Definition at line 19 of file [ASManager.hpp](#).

Referenced by [ASManager::createAccelerationStructureInfosBLAS\(\)](#).

#### 5.4.2.2 [as\\_geometry](#)

```
std::vector<VkAccelerationStructureGeometryKHR> BlasInput::as_geometry
```

Definition at line 18 of file [ASManager.hpp](#).

Referenced by [ASManager::createAccelerationStructureInfosBLAS\(\)](#).

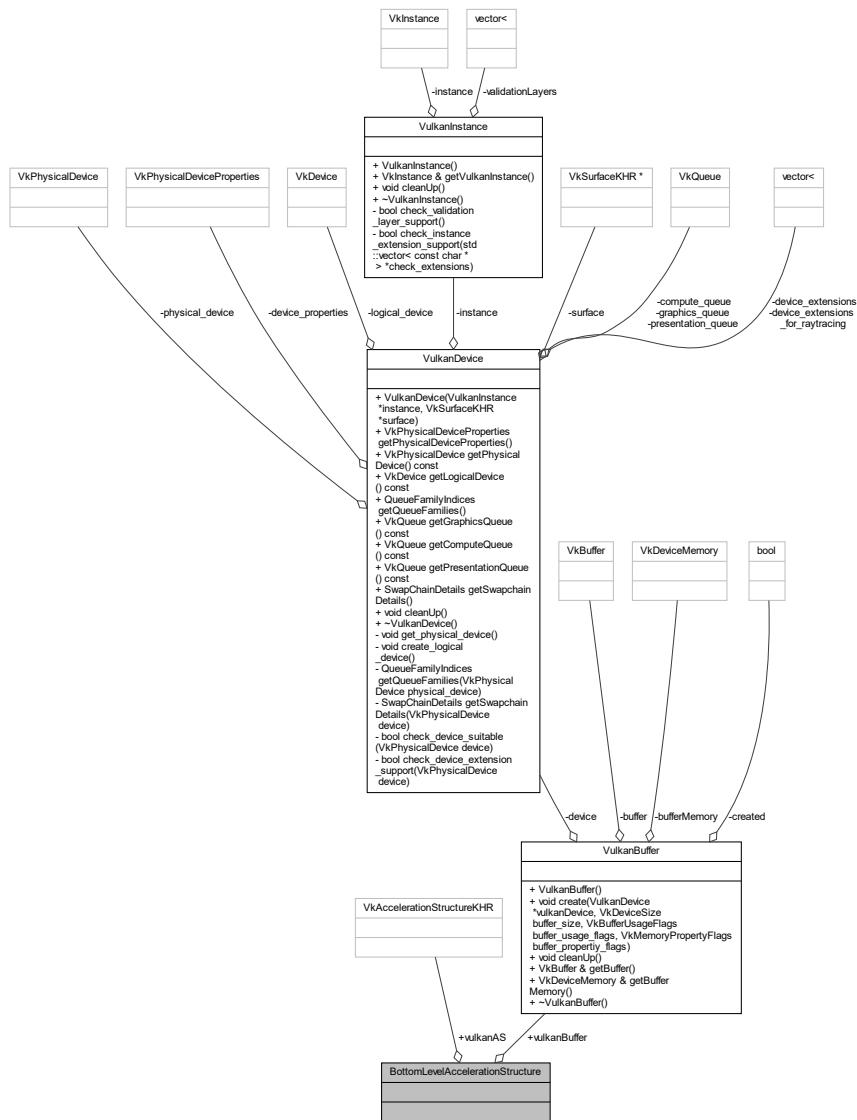
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/[ASManager.hpp](#)

## 5.5 BottomLevelAccelerationStructure Struct Reference

```
#include <BottomLevelAccelerationStructure.hpp>
```

Collaboration diagram for BottomLevelAccelerationStructure:



### Data Fields

- `VkAccelerationStructureKHR` `vulkanAS`
- `VulkanBuffer` `vulkanBuffer`

#### 5.5.1 Detailed Description

Definition at line 6 of file `BottomLevelAccelerationStructure.hpp`.

## 5.5.2 Field Documentation

### 5.5.2.1 **vulkanAS**

VkAccelerationStructureKHR BottomLevelAccelerationStructure::vulkanAS

Definition at line 7 of file [BottomLevelAccelerationStructure.hpp](#).

Referenced by [ASManager::createSingleBlas\(\)](#).

### 5.5.2.2 **vulkanBuffer**

VulkanBuffer BottomLevelAccelerationStructure::vulkanBuffer

Definition at line 8 of file [BottomLevelAccelerationStructure.hpp](#).

Referenced by [ASManager::createSingleBlas\(\)](#).

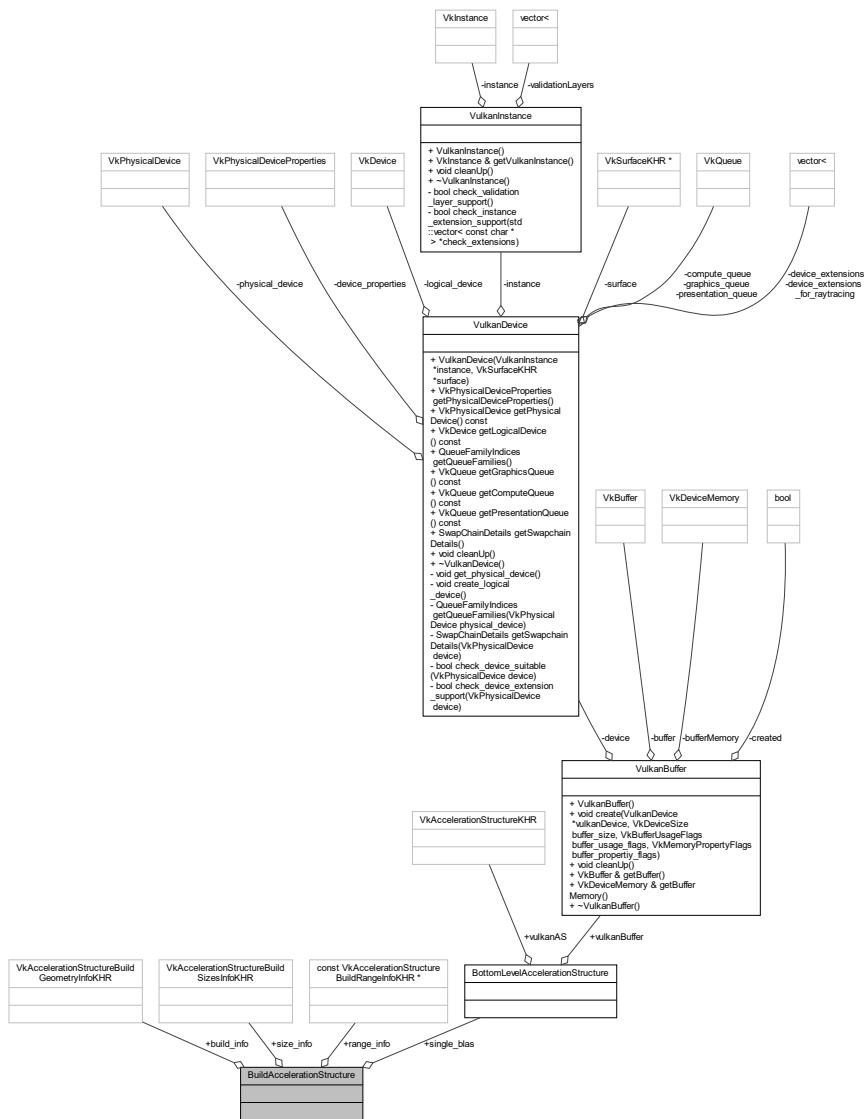
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/[BottomLevelAccelerationStructure.hpp](#)

## 5.6 BuildAccelerationStructure Struct Reference

```
#include <ASManager.hpp>
```

Collaboration diagram for BuildAccelerationStructure:



## Data Fields

- `VkAccelerationStructureBuildGeometryInfoKHR build_info`
- `VkAccelerationStructureBuildSizesInfoKHR size_info`
- `const VkAccelerationStructureBuildRangeInfoKHR * range_info`
- `BottomLevelAccelerationStructure single_blas`

### 5.6.1 Detailed Description

Definition at line 10 of file [ASManager.hpp](#).

### 5.6.2 Field Documentation

### 5.6.2.1 build\_info

```
VkAccelerationStructureBuildGeometryInfoKHR BuildAccelerationStructure::build_info
```

Definition at line 11 of file [ASManager.hpp](#).

Referenced by [ASManager::createAccelerationStructureInfosBLAS\(\)](#), and [ASManager::createSingleBlas\(\)](#).

### 5.6.2.2 range\_info

```
const VkAccelerationStructureBuildRangeInfoKHR* BuildAccelerationStructure::range_info
```

Definition at line 13 of file [ASManager.hpp](#).

Referenced by [ASManager::createAccelerationStructureInfosBLAS\(\)](#), and [ASManager::createSingleBlas\(\)](#).

### 5.6.2.3 single\_bias

```
BottomLevelAccelerationStructure BuildAccelerationStructure::single_bias
```

Definition at line 14 of file [ASManager.hpp](#).

Referenced by [ASManager::createSingleBlas\(\)](#).

### 5.6.2.4 size\_info

```
VkAccelerationStructureBuildSizesInfoKHR BuildAccelerationStructure::size_info
```

Definition at line 12 of file [ASManager.hpp](#).

Referenced by [ASManager::createAccelerationStructureInfosBLAS\(\)](#), and [ASManager::createSingleBlas\(\)](#).

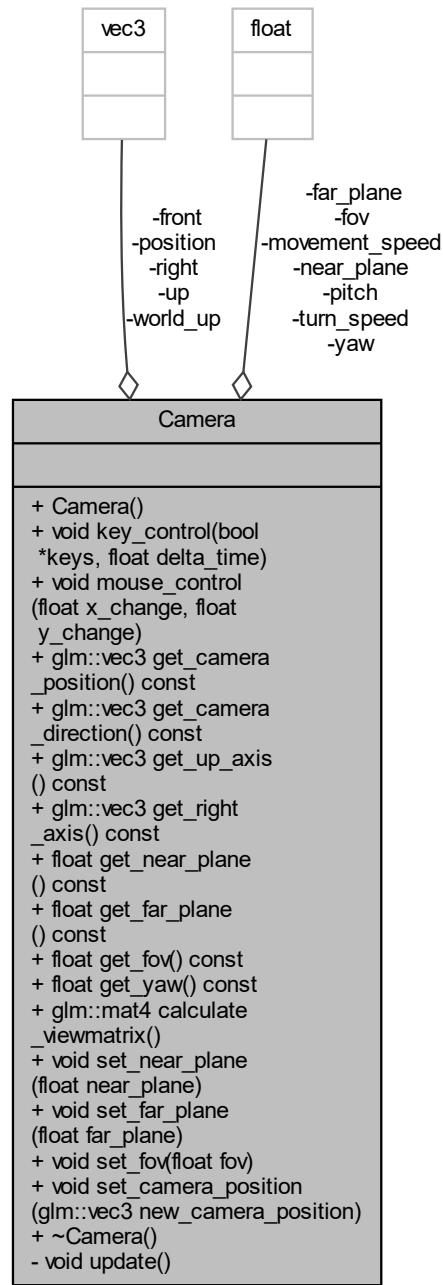
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/[ASManager.hpp](#)

## 5.7 Camera Class Reference

```
#include <Camera.hpp>
```

Collaboration diagram for Camera:



### Public Member Functions

- [Camera \(\)](#)

- void `key_control` (bool \*keys, float delta\_time)
- void `mouse_control` (float x\_change, float y\_change)
- `glm::vec3 get_camera_position () const`
- `glm::vec3 get_camera_direction () const`
- `glm::vec3 get_up_axis () const`
- `glm::vec3 get_right_axis () const`
- float `get_near_plane () const`
- float `get_far_plane () const`
- float `get_fov () const`
- float `get_yaw () const`
- `glm::mat4 calculate_viewmatrix ()`
- void `set_near_plane (float near_plane)`
- void `set_far_plane (float far_plane)`
- void `set_fov (float fov)`
- void `set_camera_position (glm::vec3 new_camera_position)`
- `~Camera ()`

## Private Member Functions

- void `update ()`

## Private Attributes

- `glm::vec3 position`
- `glm::vec3 front`
- `glm::vec3 world_up`
- `glm::vec3 right`
- `glm::vec3 up`
- float `yaw`
- float `pitch`
- float `movement_speed`
- float `turn_speed`
- float `near_plane`
- float `far_plane`
- float `fov`

### 5.7.1 Detailed Description

Definition at line 11 of file [Camera.hpp](#).

### 5.7.2 Constructor & Destructor Documentation

### 5.7.2.1 Camera()

```
Camera::Camera ( )
```

Definition at line 3 of file [Camera.cpp](#).

```
00004   :
00005
00006     position(glm::vec3(0.0f, 100.0f, -80.0f)),
00007     front(glm::vec3(0.0f, 0.0f, -1.0f)),
00008     world_up(glm::vec3(0.0f, 1.0f, 0.0f)),
00009     right(glm::normalize(glm::cross(front, world_up))),
00010     up(glm::normalize(glm::cross(right, front))),
00011     yaw(80.0f),
00012     pitch(-40.0f),
00013     movement_speed(200.0f),
00014     turn_speed(0.25f),
00015     near_plane(0.1f),
00016     far_plane(4000.0f),
00017     fov(45.0f)
00018
00019 { }
```

### 5.7.2.2 ~Camera()

```
Camera::~Camera ( )
```

Definition at line 86 of file [Camera.cpp](#).

```
00086 { }
```

## 5.7.3 Member Function Documentation

### 5.7.3.1 calculate\_viewmatrix()

```
glm::mat4 Camera::calculate_viewmatrix ( )
```

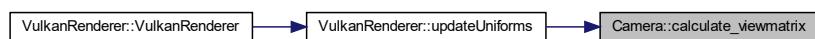
Definition at line 81 of file [Camera.cpp](#).

```
00081
00082   // very necessary for further calc
00083   return glm::lookAt(position, position + front, up);
00084 }
```

References [front](#), [position](#), and [up](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.7.3.2 get\_camera\_direction()

```
glm::vec3 Camera::get_camera_direction() const [inline]
```

Definition at line 19 of file [Camera.hpp](#).

```
00019 { return glm::normalize(front); }
```

References [front](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.7.3.3 get\_camera\_position()

```
glm::vec3 Camera::get_camera_position() const [inline]
```

Definition at line 18 of file [Camera.hpp](#).

```
00018 { return position; }
```

References [position](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.7.3.4 get\_far\_plane()

```
float Camera::get_far_plane ( ) const [inline]
```

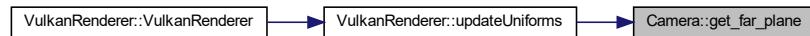
Definition at line 23 of file [Camera.hpp](#).

```
00023 { return far_plane; };
```

References [far\\_plane](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.7.3.5 get\_fov()

```
float Camera::get_fov ( ) const [inline]
```

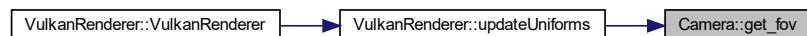
Definition at line 24 of file [Camera.hpp](#).

```
00024 { return fov; };
```

References [fov](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.7.3.6 get\_near\_plane()

```
float Camera::get_near_plane ( ) const [inline]
```

Definition at line 22 of file [Camera.hpp](#).

```
00022 { return near_plane; };
```

References [near\\_plane](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.7.3.7 get\_right\_axis()

```
glm::vec3 Camera::get_right_axis() const [inline]
```

Definition at line 21 of file [Camera.hpp](#).

```
00021 { return right; };
```

References [right](#).

### 5.7.3.8 get\_up\_axis()

```
glm::vec3 Camera::get_up_axis() const [inline]
```

Definition at line 20 of file [Camera.hpp](#).

```
00020 { return up; };
```

References [up](#).

### 5.7.3.9 get\_yaw()

```
float Camera::get_yaw() const [inline]
```

Definition at line 25 of file [Camera.hpp](#).

```
00025 { return yaw; };
```

References [yaw](#).

### 5.7.3.10 key\_control()

```
void Camera::key_control(
    bool * keys,
    float delta_time)
```

Definition at line 21 of file [Camera.cpp](#).

```
00021
00022     float velocity = movement_speed * delta_time;
00023
00024     if (keys[GLFW_KEY_W]) {
00025         position += front * velocity;
00026     }
00027
00028     if (keys[GLFW_KEY_D]) {
00029         position += right * velocity;
00030     }
00031
00032     if (keys[GLFW_KEY_A]) {
00033         position += -right * velocity;
00034     }
00035
00036     if (keys[GLFW_KEY_S]) {
00037         position += -front * velocity;
00038     }
00039
00040     if (keys[GLFW_KEY_Q]) {
00041         yaw += -velocity;
00042     }
00043
00044     if (keys[GLFW_KEY_E]) {
00045         yaw += velocity;
00046     }
00047 }
```

References [front](#), [movement\\_speed](#), [position](#), [right](#), and [yaw](#).

### 5.7.3.11 mouse\_control()

```
void Camera::mouse_control (
    float x_change,
    float y_change )
```

Definition at line 49 of file Camera.cpp.

```
00049
00050     // here we only want to support views 90 degrees to each side
00051     // again choose turn speed well in respect to its ordinal scale
00052     x_change *= turn_speed;
00053     y_change *= turn_speed;
00054
00055     yaw += x_change;
00056     pitch += y_change;
00057
00058     if (pitch > 89.0f) {
00059         pitch = 89.0f;
00060     }
00061
00062     if (pitch < -89.0f) {
00063         pitch = -89.0f;
00064     }
00065
00066     // by changing the rotations you need to update all parameters
00067     // for we retrieve them later for further calculations!
00068     update();
00069 }
```

References [pitch](#), [turn\\_speed](#), [update\(\)](#), and [yaw](#).

Here is the call graph for this function:



### 5.7.3.12 set\_camera\_position()

```
void Camera::set_camera_position (
    glm::vec3 new_camera_position )
```

Definition at line 77 of file Camera.cpp.

```
00077
00078     this->position = new_camera_position;
00079 }
```

References [position](#).

### 5.7.3.13 set\_far\_plane()

```
void Camera::set_far_plane (
    float far_plane )
```

Definition at line 73 of file [Camera.cpp](#).  
 00073 { this->far\_plane = far\_plane; }

References [far\\_plane](#).

### 5.7.3.14 set\_fov()

```
void Camera::set_fov (
    float fov )
```

Definition at line 75 of file [Camera.cpp](#).  
 00075 { this->fov = fov; }

References [fov](#).

### 5.7.3.15 set\_near\_plane()

```
void Camera::set_near_plane (
    float near_plane )
```

Definition at line 71 of file [Camera.cpp](#).  
 00071 { this->near\_plane = near\_plane; }

References [near\\_plane](#).

### 5.7.3.16 update()

```
void Camera::update ( ) [private]
```

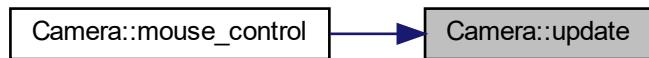
Definition at line 88 of file [Camera.cpp](#).

```
00088 {
00089     // https://learnopengl.com/Getting-started/Camera?fbclid=IwARIWER4jt6IyWC52s_WKYHtaFoeug37pG5YqbDPifgn5F1UXPbUjWbJWiQQ
00090     // thats a bit tricky; have a look to link above if there a questions :)
00091     // but simple geometrical analysis
00092     // consider yaw you are turnig to the side; pitch as you move the head forward
00093     // and back; roll rotations around z-axis will make you dizzy :)) notice that
00094     // to roll will not chnge my front vector
00095     front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
00096     front.y = sin(glm::radians(pitch));
00097     front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
00098     front = glm::normalize(front);
00099
00100    // retrieve the right vector with some world_up
00101    right = glm::normalize(glm::cross(front, world_up));
00102
00103    // but this means the up vector must again be calculated with right vector
00104    // calculated!!!
00105    up = glm::normalize(glm::cross(right, front));
00106 }
```

References [front](#), [pitch](#), [right](#), [up](#), [world\\_up](#), and [yaw](#).

Referenced by [mouse\\_control\(\)](#).

Here is the caller graph for this function:



## 5.7.4 Field Documentation

### 5.7.4.1 far\_plane

```
float Camera::far_plane [private]
```

Definition at line [49](#) of file [Camera.hpp](#).

Referenced by [get\\_far\\_plane\(\)](#), and [set\\_far\\_plane\(\)](#).

### 5.7.4.2 fov

```
float Camera::fov [private]
```

Definition at line [49](#) of file [Camera.hpp](#).

Referenced by [get\\_fov\(\)](#), and [set\\_fov\(\)](#).

### 5.7.4.3 front

```
glm::vec3 Camera::front [private]
```

Definition at line [38](#) of file [Camera.hpp](#).

Referenced by [calculate\\_viewmatrix\(\)](#), [get\\_camera\\_direction\(\)](#), [key\\_control\(\)](#), and [update\(\)](#).

#### 5.7.4.4 movement\_speed

```
float Camera::movement_speed [private]
```

Definition at line 46 of file [Camera.hpp](#).

Referenced by [key\\_control\(\)](#).

#### 5.7.4.5 near\_plane

```
float Camera::near_plane [private]
```

Definition at line 49 of file [Camera.hpp](#).

Referenced by [get\\_near\\_plane\(\)](#), and [set\\_near\\_plane\(\)](#).

#### 5.7.4.6 pitch

```
float Camera::pitch [private]
```

Definition at line 44 of file [Camera.hpp](#).

Referenced by [mouse\\_control\(\)](#), and [update\(\)](#).

#### 5.7.4.7 position

```
glm::vec3 Camera::position [private]
```

Definition at line 37 of file [Camera.hpp](#).

Referenced by [calculate\\_viewmatrix\(\)](#), [get\\_camera\\_position\(\)](#), [key\\_control\(\)](#), and [set\\_camera\\_position\(\)](#).

#### 5.7.4.8 right

```
glm::vec3 Camera::right [private]
```

Definition at line 40 of file [Camera.hpp](#).

Referenced by [get\\_right\\_axis\(\)](#), [key\\_control\(\)](#), and [update\(\)](#).

#### 5.7.4.9 turn\_speed

```
float Camera::turn_speed [private]
```

Definition at line 47 of file [Camera.hpp](#).

Referenced by [mouse\\_control\(\)](#).

#### 5.7.4.10 up

```
glm::vec3 Camera::up [private]
```

Definition at line 41 of file [Camera.hpp](#).

Referenced by [calculate\\_viewmatrix\(\)](#), [get\\_up\\_axis\(\)](#), and [update\(\)](#).

#### 5.7.4.11 world\_up

```
glm::vec3 Camera::world_up [private]
```

Definition at line 39 of file [Camera.hpp](#).

Referenced by [update\(\)](#).

#### 5.7.4.12 yaw

```
float Camera::yaw [private]
```

Definition at line 43 of file [Camera.hpp](#).

Referenced by [get\\_yaw\(\)](#), [key\\_control\(\)](#), [mouse\\_control\(\)](#), and [update\(\)](#).

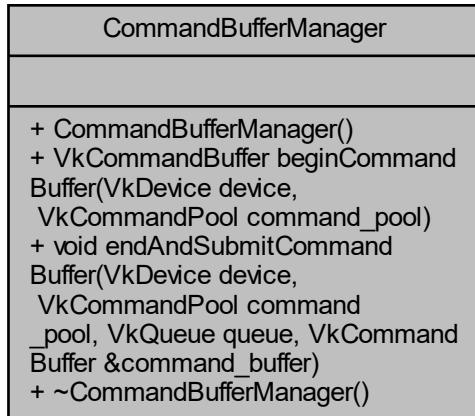
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[Camera.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/[Camera.cpp](#)

## 5.8 CommandBufferManager Class Reference

```
#include <CommandBufferManager.hpp>
```

Collaboration diagram for CommandBufferManager:



### Public Member Functions

- [CommandBufferManager \(\)](#)
- [VkCommandBuffer beginCommandBuffer \(VkDevice device, VkCommandPool command\\_pool\)](#)
- [void endAndSubmitCommandBuffer \(VkDevice device, VkCommandPool command\\_pool, VkQueue queue, VkCommandBuffer &command\\_buffer\)](#)
- [~CommandBufferManager \(\)](#)

#### 5.8.1 Detailed Description

Definition at line [6](#) of file [CommandBufferManager.hpp](#).

#### 5.8.2 Constructor & Destructor Documentation

##### 5.8.2.1 CommandBufferManager()

```
CommandBufferManager::CommandBufferManager ( )
```

Definition at line [3](#) of file [CommandBufferManager.cpp](#).  
00003 { }

### 5.8.2.2 ~CommandBufferManager()

```
CommandBufferManager::~CommandBufferManager ( )
```

Definition at line 56 of file [CommandBufferManager.cpp](#).  
00056 { }

### 5.8.3 Member Function Documentation

### 5.8.3.1 beginCommandBuffer()

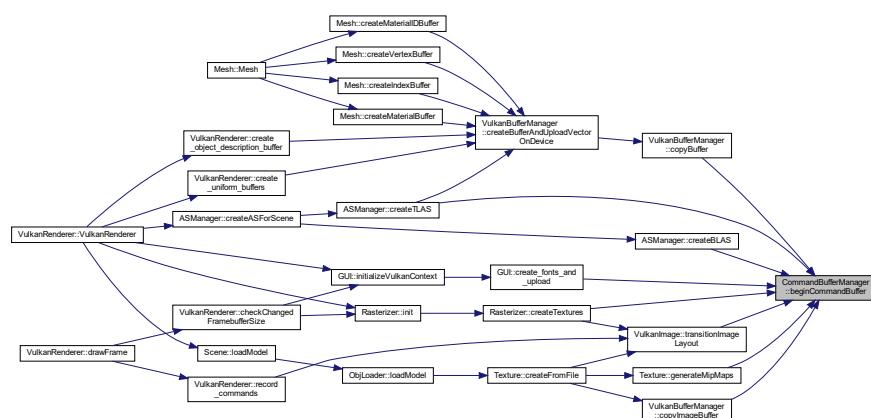
```
        VkDevice device,  
        VkCommandPool command_pool )
```

Definition at line 5 of file [CommandBufferManager.cpp](#).

```
00006
00007 // command buffer to hold transfer commands
00008 VkCommandBuffer command_buffer;
00009
00010 // command buffer details
00011 VkCommandBufferAllocateInfo alloc_info{};
00012 alloc_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
00013 alloc_info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
00014 alloc_info.commandPool = command_pool;
00015 alloc_info.commandBufferCount = 1;
00016
00017 // allocate command buffer from pool
00018 vkAllocateCommandBuffers(device, &alloc_info, &command_buffer);
00019
00020 // information to begin the command buffer record
00021 VkCommandBufferBeginInfo begin_info{};
00022 begin_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
00023 // we are only using the command buffer once, so set up for one time submit
00024 begin_info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
00025
00026 // begin recording transfer commands
00027 vkBeginCommandBuffer(command_buffer, &begin_info);
00028
00029 return command_buffer;
00030 }
```

Referenced by [VulkanBufferManager::copyBuffer\(\)](#), [VulkanBufferManager::copyImageBuffer\(\)](#), [GUI::create\\_fonts\\_and\\_upload\(\)](#), [ASManager::createBLAS\(\)](#), [Rasterizer::createTextures\(\)](#), [ASManager::createTLAS\(\)](#), [Texture::generateMipMaps\(\)](#), and [VulkanImage::transitionImageLayout\(\)](#).

Here is the caller graph for this function:



### 5.8.3.2 endAndSubmitCommandBuffer()

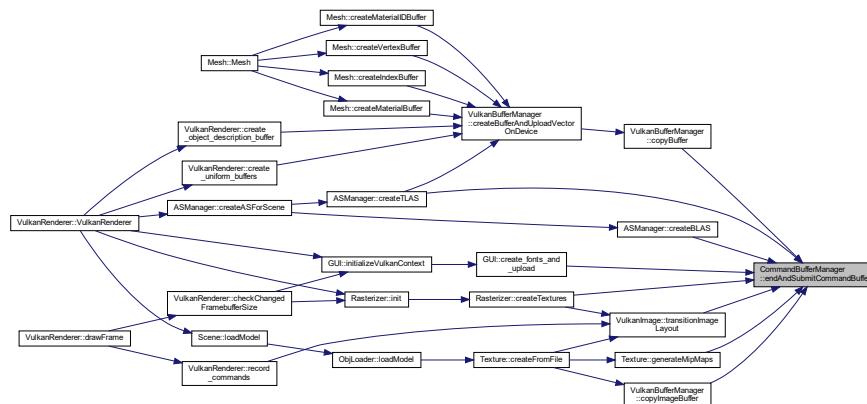
```
void CommandBufferManager::endAndSubmitCommandBuffer (
    VkDevice device,
    VkCommandPool command_pool,
    VkQueue queue,
    VkCommandBuffer & command_buffer )
```

Definition at line 32 of file [CommandBufferManager.cpp](#).

```
00034     {
00035         // end commands
00036         VkResult result = vkEndCommandBuffer(command_buffer);
00037         ASSERT_VULKAN(result, "Failed to end command buffer!")
00038
00039         // queue submission information
00040         VkSubmitInfo submit_info{};
00041         submit_info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
00042         submit_info.commandBufferCount = 1;
00043         submit_info.pCommandBuffers = &command_buffer;
00044
00045         // submit transfer command to transfer queue and wait until it finishes
00046         result = vkQueueSubmit(queue, 1, &submit_info, VK_NULL_HANDLE);
00047         ASSERT_VULKAN(result, "Failed to submit to queue!")
00048
00049         result = vkQueueWaitIdle(queue);
00050         ASSERT_VULKAN(result, "Failed to wait Idle!")
00051
00052         // free temporary command buffer back to pool
00053         vkFreeCommandBuffers(device, command_pool, 1, &command_buffer);
00054 }
```

Referenced by [VulkanBufferManager::copyBuffer\(\)](#), [VulkanBufferManager::copyImageBuffer\(\)](#), [GUI::create\\_fonts\\_and\\_upload\(\)](#), [ASManager::createBLAS\(\)](#), [Rasterizer::createTextures\(\)](#), [ASManager::createTLAS\(\)](#), [Texture::generateMipMaps\(\)](#), and [VulkanImage::transitionImageLayout\(\)](#).

Here is the caller graph for this function:



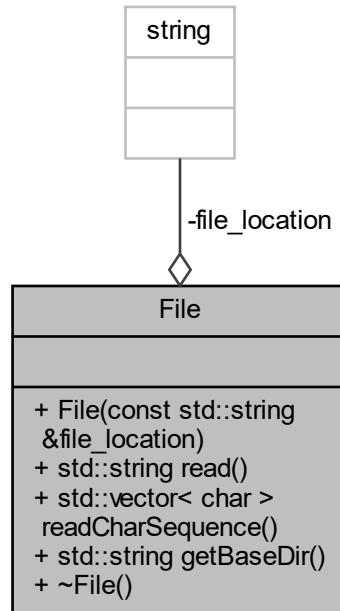
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[CommandBufferManager.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/[CommandBufferManager.cpp](#)

## 5.9 File Class Reference

```
#include <File.hpp>
```

Collaboration diagram for File:



### Public Member Functions

- `File (const std::string &file_location)`
- `std::string read ()`
- `std::vector< char > readCharSequence ()`
- `std::string getBaseDir ()`
- `~File ()`

### Private Attributes

- `std::string file_location`

#### 5.9.1 Detailed Description

Definition at line 5 of file [File.hpp](#).

#### 5.9.2 Constructor & Destructor Documentation

### 5.9.2.1 File()

```
File::File (
    const std::string & file_location ) [explicit]
```

Definition at line 6 of file [File.cpp](#).

```
00006
00007     this->file_location = file_location;
00008 }
```

References [file\\_location](#).

### 5.9.2.2 ~File()

```
File::~File ( )
```

Definition at line 60 of file [File.cpp](#).

```
00060 { }
```

## 5.9.3 Member Function Documentation

### 5.9.3.1 getBaseDir()

```
std::string File::getBaseDir ( )
```

Definition at line 54 of file [File.cpp](#).

```
00054     {
00055         if (file_location.find_last_of("//\\") != std::string::npos)
00056             return file_location.substr(0, file_location.find_last_of("//\\"));
00057         return "";
00058     }
```

References [file\\_location](#).

Referenced by [ObjLoader::loadTexturesAndMaterials\(\)](#).

Here is the caller graph for this function:



### 5.9.3.2 `read()`

```
std::string File::read ( )
```

**Definition at line 10 of file [File.cpp](#).**

```
00010     {
00011     std::string content;
00012     std::ifstream file_stream(file_location, std::ios::in);
00013
00014     if (!file_stream.is_open()) {
00015         printf("Failed to read %. File does not exist.", file_location.c_str());
00016         return "";
00017     }
00018
00019     std::string line = "";
00020     while (!file_stream.eof()) {
00021         std::getline(file_stream, line);
00022         content.append(line + "\n");
00023     }
00024
00025     file_stream.close();
00026     return content;
00027 }
```

References [file\\_location](#).

### 5.9.3.3 `readCharSequence()`

```
std::vector< char > File::readCharSequence ( )
```

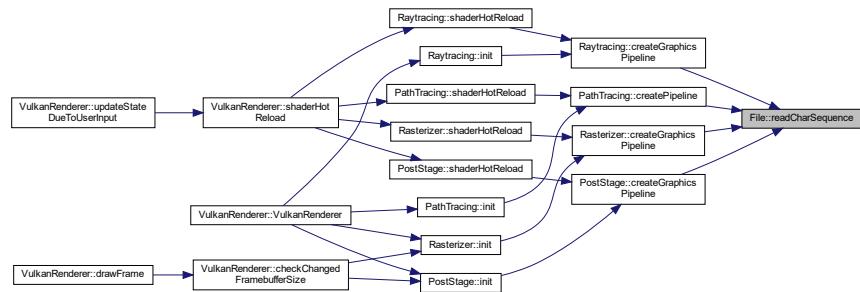
**Definition at line 29 of file [File.cpp](#).**

```
00029     {
00030     // open stream from given file
00031     // std::ios::binary tells stream to read file as binary
00032     // std::ios::ate tells stream to start reading from end of file
00033     std::ifstream file(file_location, std::ios::binary | std::ios::ate);
00034
00035     // check if file stream sucessfully opened
00036     if (!file.is_open()) {
00037         throw std::runtime_error("Failed to open a file!");
00038     }
00039
00040     size_t file_size = (size_t)file.tellg();
00041     std::vector<char> file_buffer(file_size);
00042
00043     // move read position to start of file
00044     file.seekg(0);
00045
00046     // read the file data into the buffer (stream "file_size" in total)
00047     file.read(file_buffer.data(), file_size);
00048
00049     file.close();
00050
00051     return file_buffer;
00052 }
```

References [file\\_location](#).

Referenced by [PostStage::createGraphicsPipeline\(\)](#), [Rasterizer::createGraphicsPipeline\(\)](#), [Raytracing::createGraphicsPipeline\(\)](#), and [PathTracing::createPipeline\(\)](#).

Here is the caller graph for this function:



## 5.9.4 Field Documentation

### 5.9.4.1 file\_location

```
std::string File::file_location [private]
```

Definition at line 16 of file [File.hpp](#).

Referenced by [File\(\)](#), [getBaseDir\(\)](#), [read\(\)](#), and [readCharSequence\(\)](#).

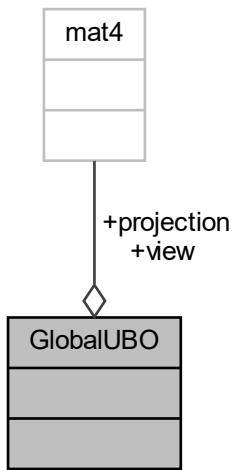
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/util/[File.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/util/[File.cpp](#)

## 5.10 GlobalUBO Struct Reference

```
#include <GlobalUBO.hpp>
```

Collaboration diagram for GlobalUBO:



## Data Fields

- `mat4 projection`
- `mat4 view`

### 5.10.1 Detailed Description

Definition at line 18 of file [GlobalUBO.hpp](#).

### 5.10.2 Field Documentation

#### 5.10.2.1 `projection`

`mat4 GlobalUBO::projection`

Definition at line 19 of file [GlobalUBO.hpp](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

### 5.10.2.2 view

`mat4 GlobalUBO::view`

Definition at line 20 of file [GlobalUBO.hpp](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

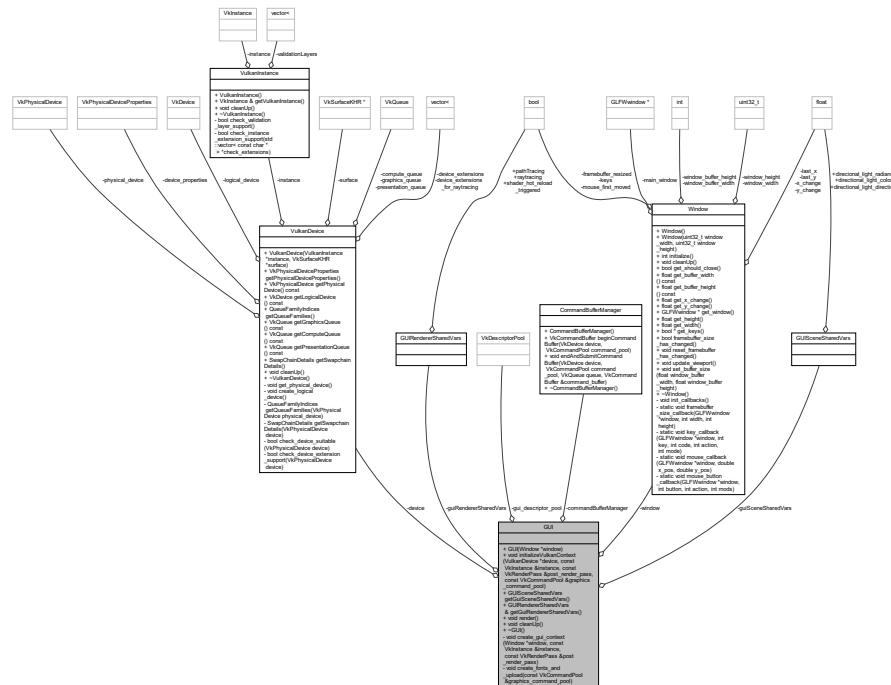
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/GlobalUBO.hpp

## 5.11 GUI Class Reference

```
#include <GUI.hpp>
```

Collaboration diagram for GUI:



## Public Member Functions

- `GUI (Window *window)`
- `void initializeVulkanContext (VulkanDevice *device, const VkInstance &instance, const VkRenderPass &post_render_pass, const VkCommandPool &graphics_command_pool)`
- `GUISceneSharedVars getGuiSceneSharedVars ()`
- `GUIRendererSharedVars & getGuiRendererSharedVars ()`
- `void render ()`
- `void cleanUp ()`
- `~GUI ()`

## Private Member Functions

- void `create_gui_context` (`Window *window`, const `VkInstance &instance`, const `VkRenderPass &post_render_pass`)
- void `create_fonts_and_upload` (const `VkCommandPool &graphics_command_pool`)

## Private Attributes

- `VulkanDevice * device` {VK\_NULL\_HANDLE}
- `Window * window` {VK\_NULL\_HANDLE}
- `VkDescriptorPool gui_descriptor_pool` {VK\_NULL\_HANDLE}
- `CommandBufferManager commandBufferManager`
- `GUISceneSharedVars guiSceneSharedVars`
- `GUIRendererSharedVars guiRendererSharedVars`

### 5.11.1 Detailed Description

Definition at line 18 of file `GUI.hpp`.

### 5.11.2 Constructor & Destructor Documentation

#### 5.11.2.1 `GUI()`

```
GUI::GUI (
```

```
          Window * window )
```

Definition at line 11 of file `GUI.cpp`.  
00011 { this->window = window; }

References `window`.

#### 5.11.2.2 `~GUI()`

```
GUI::~GUI ( )
```

Definition at line 240 of file `GUI.cpp`.  
00240 { }

### 5.11.3 Member Function Documentation

### 5.11.3.1 cleanUp()

```
void GUI::cleanUp ( )
```

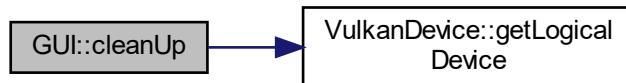
Definition at line 121 of file [GUI.cpp](#).

```
00121     {
00122     // clean up of GUI stuff
00123     ImGui_ImplVulkan_Shutdown();
00124     ImGui_ImplGlfw_Shutdown();
00125     ImGui::DestroyContext();
00126     vkDestroyDescriptorPool(device->getLogicalDevice(), gui_descriptor_pool,
00127                             nullptr);
00128 }
```

References [device](#), [VulkanDevice::getLogicalDevice\(\)](#), and [gui\\_descriptor\\_pool](#).

Referenced by [VulkanRenderer::checkChangedFrameBufferSize\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.11.3.2 create\_fonts\_and\_upload()

```
void GUI::create_fonts_and_upload (
    const VkCommandPool & graphics_command_pool ) [private]
```

Definition at line 226 of file [GUI.cpp](#).

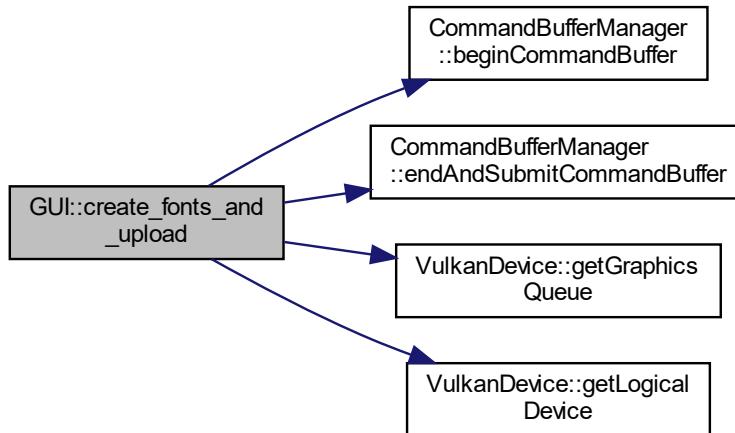
```
00226
00227     VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
00228         device->getLogicalDevice(), graphics_command_pool);
00229     ImGui_ImplVulkan_CreateFontsTexture(command_buffer);
00230     commandBufferManager.endAndSubmitCommandBuffer(
00231         device->getLogicalDevice(), graphics_command_pool,
00232         device->getGraphicsQueue(), command_buffer);
00233
00234     // wait until no actions being run on device before destroying
00235     vkDeviceWaitIdle(device->getLogicalDevice());
00236     // clear font textures from cpu data
```

```
00237     ImGui_ImplVulkan_DestroyFontUploadObjects();
00238 }
```

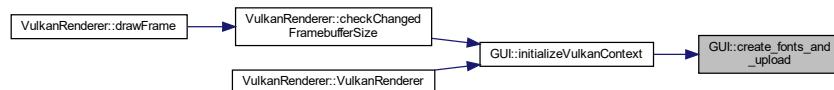
References [CommandBufferManager::beginCommandBuffer\(\)](#), [commandBufferManager](#), [device](#), [CommandBufferManager::endAndSubmitCommandBuffer\(\)](#), [VulkanDevice::getGraphicsQueue\(\)](#), and [VulkanDevice::getLogicalDevice\(\)](#).

Referenced by [initializeVulkanContext\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.11.3.3 create\_gui\_context()

```
void GUI::create_gui_context (
    Window * window,
    const VkInstance & instance,
    const VkRenderPass & post_render_pass ) [private]
```

Definition at line 130 of file [GUI.cpp](#).

```
00131
00132     IMGUI_CHECKVERSION();
00133     ImGui::CreateContext();
00134     ImGuiIO& io = ImGui::GetIO();
00135     (void)io;
00136 }
```

```

00137     float size_pixels = 18;
00138
00139     std::stringstream fontDir;
00140     std::filesystem::path cwd = std::filesystem::current_path();
00141     fontDir << cwd.string();
00142     fontDir << RELATIVE_IMGUI_FONTS_PATH;
00143
00144     std::stringstream robo_font;
00145     robo_font << fontDir.str() << "Roboto-Medium.ttf";
00146     std::stringstream Cousine_font;
00147     Cousine_font << fontDir.str() << "Cousine-Regular.ttf";
00148     std::stringstream DroidSans_font;
00149     DroidSans_font << fontDir.str() << "DroidSans.ttf";
00150     std::stringstream Karla_font;
00151     Karla_font << fontDir.str() << "Karla-Regular.ttf";
00152     std::stringstream proggy_clean_font;
00153     proggy_clean_font << fontDir.str() << "ProggyClean.ttf";
00154     std::stringstream proggy_tiny_font;
00155     proggy_tiny_font << fontDir.str() << "ProggyTiny.ttf";
00156
00157     io.Fnts->AddFontFromFileTTF(robo_font.str().c_str(), size_pixels);
00158     io.Fnts->AddFontFromFileTTF(Cousine_font.str().c_str(), size_pixels);
00159     io.Fnts->AddFontFromFileTTF(DroidSans_font.str().c_str(), size_pixels);
00160     io.Fnts->AddFontFromFileTTF(Karla_font.str().c_str(), size_pixels);
00161     io.Fnts->AddFontFromFileTTF(proggy_clean_font.str().c_str(), size_pixels);
00162     io.Fnts->AddFontFromFileTTF(proggy_tiny_font.str().c_str(), size_pixels);
00163
00164     ImGui::PushStyleVar(ImGuiStyleVar_WindowRounding, 10);
00165     ImGui::PushStyleVar(ImGuiStyleVar_FrameRounding, 10);
00166     ImGui::PushStyleVar(ImGuiStyleVar_FrameBorderSize, 1);
00167     io.ConfigFlags |=
00168         ImGuiConfigFlags_NavEnableKeyboard; // Enable Keyboard Controls
00169     io.ConfigFlags |= ImGuiConfigFlags_NavEnableSetMousePos;
00170     io.WantCaptureMouse = true;
00171     // io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad; // Enable Gamepad
00172     // Controls
00173
00174     // Setup Dear ImGui style
00175     ImGui::StyleColorsDark();
00176     // ImGui::StyleColorsClassic();
00177
00178     ImGui_ImplGlfw_InitForVulkan(window->get_window(), false);
00179
00180     // Create Descriptor Pool
00181     VkDescriptorPoolSize gui_pool_sizes[] = {
00182         {VK_DESCRIPTOR_TYPE_SAMPLER, 10},
00183         {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 10},
00184         {VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 10},
00185         {VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 10},
00186         {VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, 10},
00187         {VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, 10},
00188         {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 10},
00189         {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 10},
00190         {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 10},
00191         {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, 10},
00192         {VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, 100}};
00193
00194     VkDescriptorPoolCreateInfo gui_pool_info = {};
00195     gui_pool_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00196     gui_pool_info.flags = VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT;
00197     gui_pool_info.maxSets = 10 * IM_ARRAYSIZE(gui_pool_sizes);
00198     gui_pool_info.poolSizeCount = (uint32_t)IM_ARRAYSIZE(gui_pool_sizes);
00199     gui_pool_info.pPoolSizes = gui_pool_sizes;
00200
00201     VkResult result =
00202         vkCreateDescriptorPool(device->getLogicalDevice(), &gui_pool_info,
00203             nullptr, &gui_descriptor_pool);
00204     ASSERT_VULKAN(result, "Failed to create a gui descriptor pool!")
00205
00206     QueueFamilyIndices indices = device->getQueueFamilies();
00207
00208     ImGui_ImplVulkan_InitInfo init_info = {};
00209     init_info.Instance = instance;
00210     init_info.PhysicalDevice = device->getPhysicalDevice();
00211     init_info.Device = device->getLogicalDevice();
00212     init_info.QueueFamily = indices.graphics_family;
00213     init_info.Queue = device->getGraphicsQueue();
00214     init_info.DescriptorPool = gui_descriptor_pool;
00215     init_info.PipelineCache = VK_NULL_HANDLE;
00216     init_info.MinImageCount = 2;
00217     init_info.ImageCount = MAX_FRAME_DRAWS;
00218     init_infoAllocator = VK_NULL_HANDLE;
00219     init_info.CheckVkResultFn = VK_NULL_HANDLE;
00220     init_info.Subpass = 0;
00221     init_info.MSAASamples = VK_SAMPLE_COUNT_1_BIT;
00222
00223     ImGui_ImplVulkan_Init(&init_info, post_render_pass);

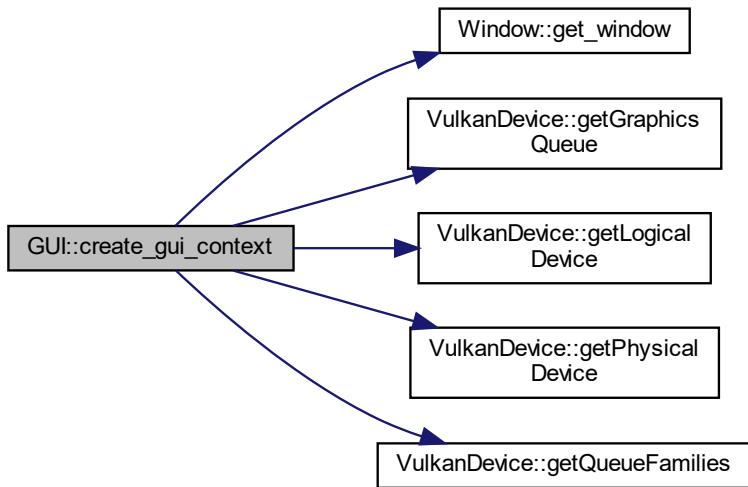
```

```
00224 }
```

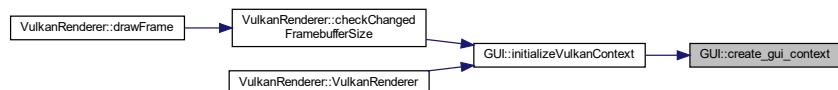
References `device`, `Window::get_window()`, `VulkanDevice::getGraphicsQueue()`, `VulkanDevice::getLogicalDevice()`, `VulkanDevice::getPhysicalDevice()`, `VulkanDevice::getQueueFamilies()`, `QueueFamilyIndices::graphics_family`, `gui_descriptor_pool`, `MAX_FRAME_DRAWs`, and `window`.

Referenced by `initializeVulkanContext()`.

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.11.3.4 getGuiRendererSharedVars()

```
GUIRendererSharedVars & GUI::getGuiRendererSharedVars( ) [inline]
```

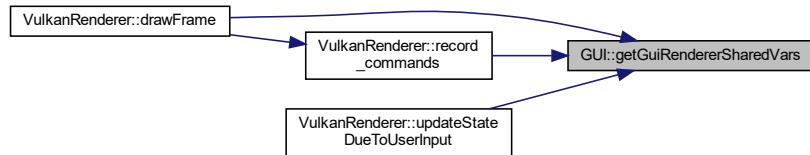
Definition at line 27 of file `GUI.hpp`.

```
00027
00028     return guiRendererSharedVars;
00029 }
```

References `guiRendererSharedVars`.

Referenced by [VulkanRenderer::drawFrame\(\)](#), [VulkanRenderer::record\\_commands\(\)](#), and [VulkanRenderer::updateStateDueToUserInput\(\)](#)

Here is the caller graph for this function:



### 5.11.3.5 getGuiSceneSharedVars()

[GUISceneSharedVars](#) GUI::getGuiSceneSharedVars ( ) [inline]

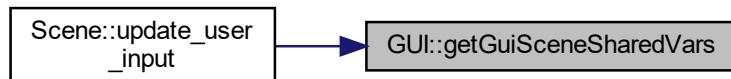
Definition at line 26 of file [GUI.hpp](#).

```
00026 { return guiSceneSharedVars; };
```

References [guiSceneSharedVars](#).

Referenced by [Scene::update\\_user\\_input\(\)](#).

Here is the caller graph for this function:



### 5.11.3.6 initializeVulkanContext()

```
void GUI::initializeVulkanContext (
    VulkanDevice * device,
    const VkInstance & instance,
    const VkRenderPass & post_render_pass,
    const VkCommandPool & graphics_command_pool )
```

Definition at line 13 of file [GUI.cpp](#).

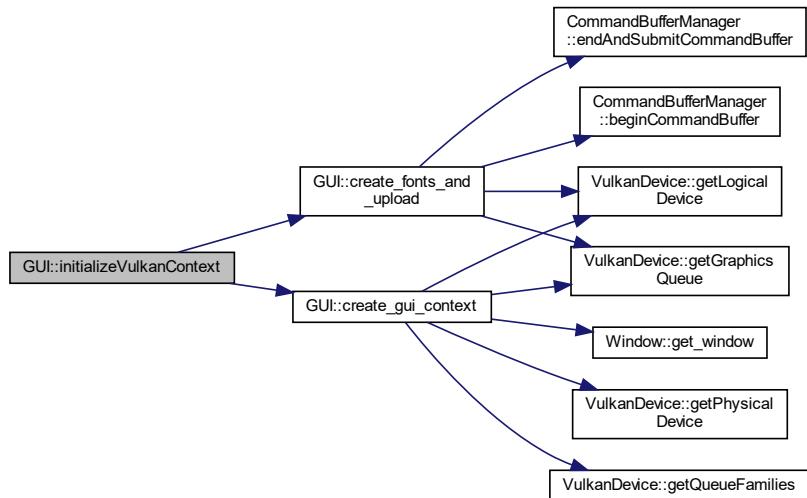
```
00016
00017     this->device = device;
00018 }
```

```
00019     create_gui_context(window, instance, post_render_pass);
00020     create_fonts_and_upload(graphics_command_pool);
00021 }
```

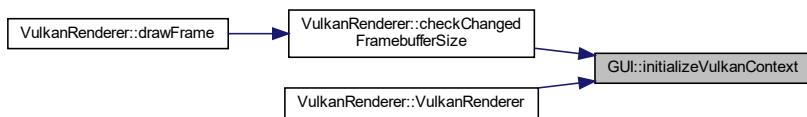
References [create\\_fonts\\_and\\_upload\(\)](#), [create\\_gui\\_context\(\)](#), [device](#), and [window](#).

Referenced by [VulkanRenderer::checkChangedFramebufferSize\(\)](#), and [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.11.3.7 render()

```
void GUI::render( )
```

Definition at line 23 of file [GUI.cpp](#).

```
00023     {
00024     // Start the Dear ImGui frame
00025     ImGui_ImplVulkan_NewFrame();
00026     ImGui_ImplGlfw_NewFrame();
00027     ImGui::NewFrame();
00028
00029     // ImGui::ShowDemoWindow();
00030 }
```

```

00031 // render your GUI
00032 ImGui::Begin("GUI v1.4.4");
00033
00034 if (ImGui::CollapsingHeader("Hot shader reload")) {
00035     if (ImGui::Button("All shader!")) {
00036         guiRendererSharedVars.shader_hot_reload_triggered = true;
00037     }
00038 }
00039
00040 ImGui::Separator();
00041
00042 static int e = 0;
00043 ImGui::RadioButton("Rasterizer", &e, 0);
00044 ImGui::SameLine();
00045 ImGui::RadioButton("Raytracing", &e, 1);
00046 ImGui::SameLine();
00047 ImGui::RadioButton("Path tracing", &e, 2);
00048
00049 switch (e) {
00050     case 0:
00051         guiRendererSharedVars.raytracing = false;
00052         guiRendererSharedVars.pathTracing = false;
00053         break;
00054     case 1:
00055         guiRendererSharedVars.raytracing = true;
00056         guiRendererSharedVars.pathTracing = false;
00057         break;
00058     case 2:
00059         guiRendererSharedVars.raytracing = false;
00060         guiRendererSharedVars.pathTracing = true;
00061         break;
00062 }
00063 // ImGui::Checkbox("Ray tracing", &guiRendererSharedVars.raytracing);
00064
00065 ImGui::Separator();
00066
00067 if (ImGui::CollapsingHeader("Graphic Settings")) {
00068     if (ImGui::TreeNode("Directional Light")) {
00069         ImGui::Separator();
00070         ImGui::SliderFloat("Ambient intensity",
00071                            &guiSceneSharedVars.direccional_light_radiance, 0.0f,
00072                            50.0f);
00073         ImGui::Separator();
00074         // Edit a color (stored as ~4 floats)
00075         ImGui::ColorEdit3("Directional Light Color",
00076                           guiSceneSharedVars.directional_light_color);
00077         ImGui::Separator();
00078         ImGui::SliderFloat3("Light Direction",
00079                            guiSceneSharedVars.directional_light_direction, -1.f,
00080                            1.0f);
00081
00082         ImGui::TreePop();
00083     }
00084 }
00085
00086 ImGui::Separator();
00087
00088 if (ImGui::CollapsingHeader("GUI Settings")) {
00089     ImGuiStyle& style = ImGui::GetStyle();
00090
00091     if (ImGui::SliderFloat("Frame Rounding", &style.FrameRounding, 0.0f, 12.0f,
00092                             "% .0f")) {
00093         style.GrabRounding = style.FrameRounding; // Make GrabRounding always the
00094                                         // same value as FrameRounding
00095     }
00096     {
00097         bool border = (style.FrameBorderSize > 0.0f);
00098         if (ImGui::Checkbox("FrameBorder", &border)) {
00099             style.FrameBorderSize = border ? 1.0f : 0.0f;
00100         }
00101     }
00102     ImGui::SliderFloat("WindowRounding", &style.WindowRounding, 0.0f, 12.0f,
00103                         "% .0f");
00104 }
00105
00106 ImGui::Separator();
00107
00108 if (ImGui::CollapsingHeader("KEY Bindings")) {
00109     ImGui::Text(
00110         "WASD for moving Forward, backward and to the side\n QE for rotating ");
00111 }
00112
00113 ImGui::Separator();
00114
00115 ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
00116              1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
00117

```

```
00118     ImGui::End();  
00119 }
```

References [GUISceneSharedVars::direcional\\_light\\_radiance](#), [GUISceneSharedVars::directional\\_light\\_color](#), [GUISceneSharedVars::directional\\_light\\_direction](#), [guiRendererSharedVars](#), [guiSceneSharedVars](#), [GUIRendererSharedVars::pathTrace](#), [GUIRendererSharedVars::raytracing](#), and [GUIRendererSharedVars::shader\\_hot\\_reload\\_triggered](#).

## 5.11.4 Field Documentation

### 5.11.4.1 commandBufferManager

```
CommandBufferManager GUI::commandBufferManager [private]
```

Definition at line [46](#) of file [GUI.hpp](#).

Referenced by [create\\_fonts\\_and\\_upload\(\)](#).

### 5.11.4.2 device

```
VulkanDevice* GUI::device {VK_NULL_HANDLE} [private]
```

Definition at line [43](#) of file [GUI.hpp](#).

Referenced by [cleanUp\(\)](#), [create\\_fonts\\_and\\_upload\(\)](#), [create\\_gui\\_context\(\)](#), and [initializeVulkanContext\(\)](#).

### 5.11.4.3 gui\_descriptor\_pool

```
VkDescriptorPool GUI::gui_descriptor_pool {VK_NULL_HANDLE} [private]
```

Definition at line [45](#) of file [GUI.hpp](#).

Referenced by [cleanUp\(\)](#), and [create\\_gui\\_context\(\)](#).

### 5.11.4.4 guiRendererSharedVars

```
GUIRendererSharedVars GUI::guiRendererSharedVars [private]
```

Definition at line [49](#) of file [GUI.hpp](#).

Referenced by [getGuiRendererSharedVars\(\)](#), and [render\(\)](#).

#### 5.11.4.5 guiSceneSharedVars

`GUI::guiSceneSharedVars` [private]

Definition at line 48 of file [GUI.hpp](#).

Referenced by [getGuiSceneSharedVars\(\)](#), and [render\(\)](#).

#### 5.11.4.6 window

`Window* GUI::window {VK_NULL_HANDLE}` [private]

Definition at line 44 of file [GUI.hpp](#).

Referenced by [create\\_gui\\_context\(\)](#), [GUI\(\)](#), and [initializeVulkanContext\(\)](#).

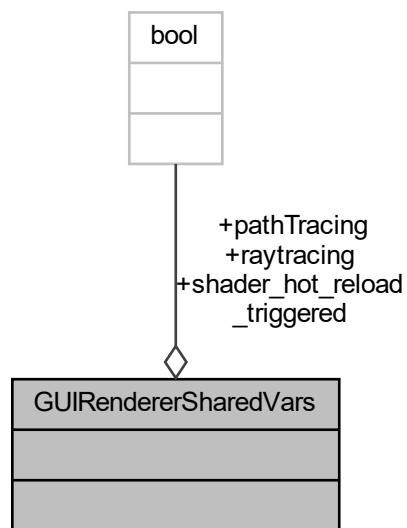
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/gui/[GUI.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/gui/[GUI.cpp](#)

## 5.12 GUIRendererSharedVars Struct Reference

#include <GUIRendererSharedVars.hpp>

Collaboration diagram for GUIRendererSharedVars:



## Data Fields

- bool `raytracing` = false
- bool `pathTracing` = false
- bool `shader_hot_reload_triggered` = false

### 5.12.1 Detailed Description

Definition at line 1 of file [GUIRendererSharedVars.hpp](#).

### 5.12.2 Field Documentation

#### 5.12.2.1 pathTracing

```
bool GUIRendererSharedVars::pathTracing = false
```

Definition at line 3 of file [GUIRendererSharedVars.hpp](#).

Referenced by [VulkanRenderer::record\\_commands\(\)](#), and [GUI::render\(\)](#).

#### 5.12.2.2 raytracing

```
bool GUIRendererSharedVars::raytracing = false
```

Definition at line 2 of file [GUIRendererSharedVars.hpp](#).

Referenced by [VulkanRenderer::drawFrame\(\)](#), [VulkanRenderer::record\\_commands\(\)](#), and [GUI::render\(\)](#).

#### 5.12.2.3 shader\_hot\_reload\_triggered

```
bool GUIRendererSharedVars::shader_hot_reload_triggered = false
```

Definition at line 5 of file [GUIRendererSharedVars.hpp](#).

Referenced by [GUI::render\(\)](#), and [VulkanRenderer::updateStateDueToUserInput\(\)](#).

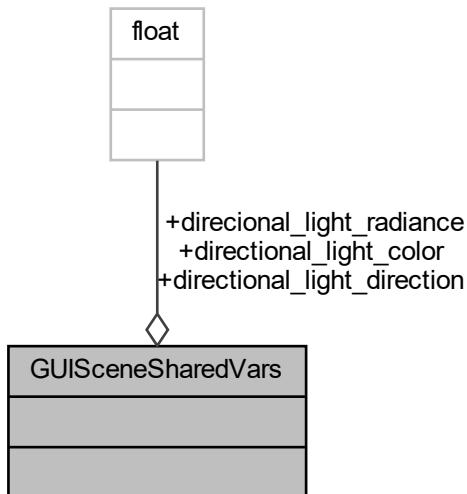
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[GUIRendererSharedVars.hpp](#)

## 5.13 GUISceneSharedVars Struct Reference

```
#include <GUISceneSharedVars.hpp>
```

Collaboration diagram for GUISceneSharedVars:



### Data Fields

- float `direccional_light_radiance` = 10.f
- float `directional_light_color` [3] = {1.f, 1.f, 1.f}
- float `directional_light_direction` [3] = {0.075f, -1.f, 0.118f}

#### 5.13.1 Detailed Description

Definition at line 2 of file [GUISceneSharedVars.hpp](#).

#### 5.13.2 Field Documentation

##### 5.13.2.1 `direccional_light_radiance`

```
float GUISceneSharedVars::direccional_light_radiance = 10.f
```

Definition at line 3 of file [GUISceneSharedVars.hpp](#).

Referenced by [GUI::render\(\)](#).

### 5.13.2.2 directional\_light\_color

```
float GUISceneSharedVars::directional_light_color[3] = {1.f, 1.f, 1.f}
```

Definition at line 4 of file [GUISceneSharedVars.hpp](#).

Referenced by [GUI::render\(\)](#).

### 5.13.2.3 directional\_light\_direction

```
float GUISceneSharedVars::directional_light_direction[3] = {0.075f, -1.f, 0.118f}
```

Definition at line 5 of file [GUISceneSharedVars.hpp](#).

Referenced by [GUI::render\(\)](#), and [VulkanRenderer::updateUniforms\(\)](#).

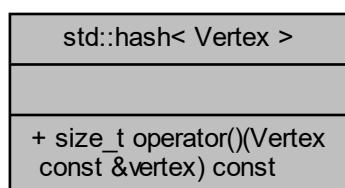
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[GUISceneSharedVars.hpp](#)

## 5.14 std::hash< Vertex > Struct Reference

```
#include <Vertex.hpp>
```

Collaboration diagram for std::hash< Vertex >:



### Public Member Functions

- `size_t operator() (Vertex const &vertex) const`

### 5.14.1 Detailed Description

Definition at line 40 of file [Vertex.hpp](#).

## 5.14.2 Member Function Documentation

### 5.14.2.1 operator()()

```
size_t std::hash< Vertex >::operator() (
    Vertex const & vertex ) const [inline]
```

Definition at line 41 of file [Vertex.hpp](#).

```
00041     {
00042         size_t h1 = hash<glm::vec3>()(vertex.pos);
00043         size_t h2 = hash<glm::vec3>()(vertex.color);
00044         size_t h3 = hash<glm::vec2>()(vertex.texture_coords);
00045         size_t h4 = hash<glm::vec3>()(vertex.normal);
00046
00047         return (((((h2 << 1) ^ h1) >> 1) ^ h3) << 1) ^ h4));
00048     }
```

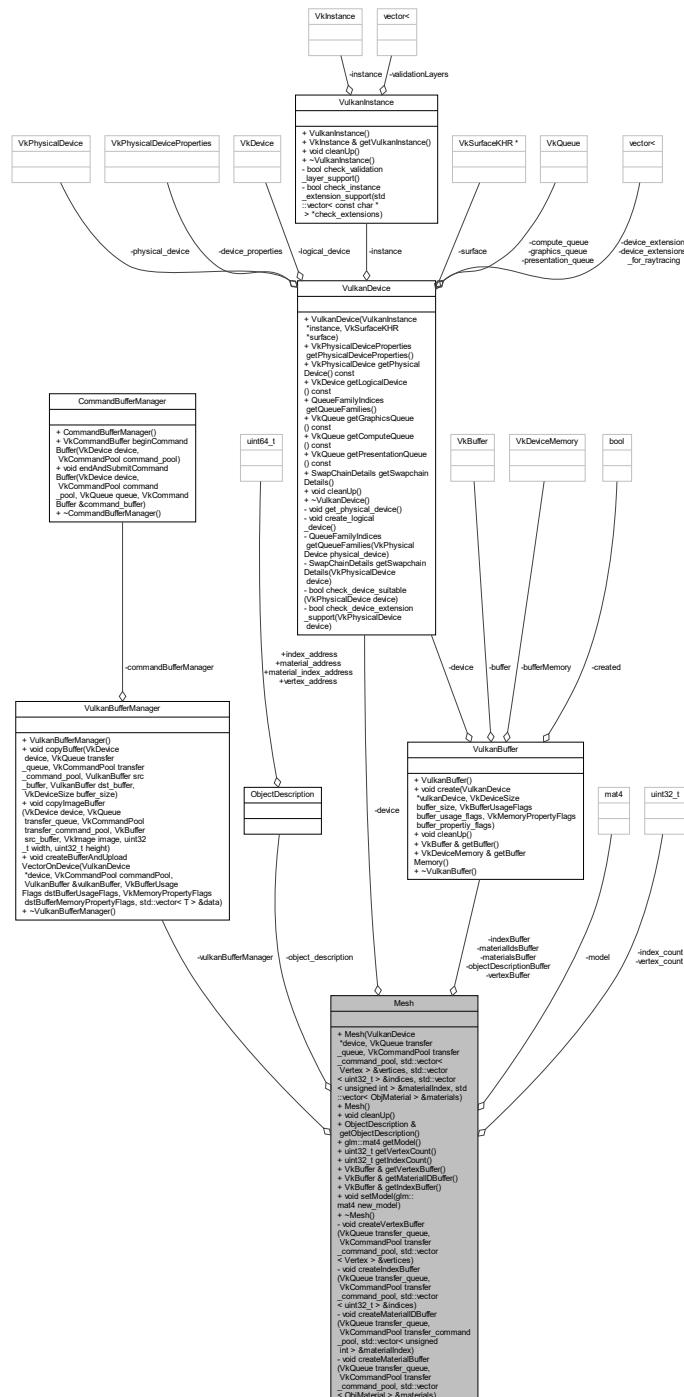
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[Vertex.hpp](#)

## 5.15 Mesh Class Reference

```
#include <Mesh.hpp>
```

## Collaboration diagram for Mesh:



## Public Member Functions

- `Mesh` (`VulkanDevice *device`, `VkQueue transfer_queue`, `VkCommandPool transfer_command_pool`, `std::vector<Vertex> &vertices`, `std::vector<uint32_t> &indices`, `std::vector<unsigned int> &materialIndex`, `std::vector<ObjMaterial> &materials`)
  - `Mesh()`
  - `void cleanUp()`

- `ObjectDescription & getObjectDescription ()`
- `glm::mat4 getModel ()`
- `uint32_t getVertexCount ()`
- `uint32_t getIndexCount ()`
- `VkBuffer & getVertexBuffer ()`
- `VkBuffer & getMaterialIDBuffer ()`
- `VkBuffer & getIndexBuffer ()`
- `void setModel (glm::mat4 new_model)`
- `~Mesh ()`

## Private Member Functions

- `void createVertexBuffer (VkQueue transfer_queue, VkCommandPool transfer_command_pool, std::vector< Vertex > &vertices)`
- `void createIndexBuffer (VkQueue transfer_queue, VkCommandPool transfer_command_pool, std::vector< uint32_t > &indices)`
- `void createMaterialIDBuffer (VkQueue transfer_queue, VkCommandPool transfer_command_pool, std::vector< unsigned int > &materialIndex)`
- `void createMaterialBuffer (VkQueue transfer_queue, VkCommandPool transfer_command_pool, std::vector< ObjMaterial > &materials)`

## Private Attributes

- `VulkanBufferManager vulkanBufferManager`
- `ObjectDescription object_description`
- `VulkanBuffer vertexBuffer`
- `VulkanBuffer indexBuffer`
- `VulkanBuffer objectDescriptionBuffer`
- `VulkanBuffer materialIdsBuffer`
- `VulkanBuffer materialsBuffer`
- `glm::mat4 model`
- `uint32_t vertex_count {static_cast<uint32_t>(-1)}`
- `uint32_t index_count {static_cast<uint32_t>(-1)}`
- `VulkanDevice * device {VK_NULL_HANDLE}`

### 5.15.1 Detailed Description

Definition at line 12 of file [Mesh.hpp](#).

### 5.15.2 Constructor & Destructor Documentation

### 5.15.2.1 Mesh() [1/2]

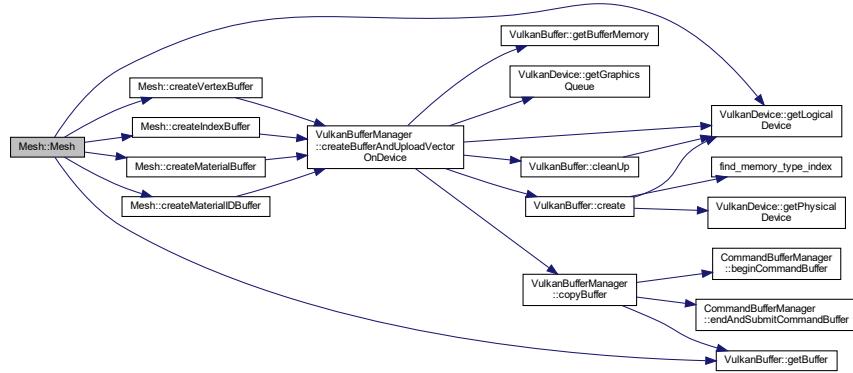
```
Mesh::Mesh (
    VulkanDevice * device,
    VkQueue transfer_queue,
    VkCommandPool transfer_command_pool,
    std::vector< Vertex > & vertices,
    std::vector< uint32_t > & indices,
    std::vector< unsigned int > & materialIndex,
    std::vector< ObjMaterial > & materials )
```

Definition at line 18 of file [Mesh.cpp](#).

```
00022                                     {
00023     // glm uses column major matrices so transpose it for Vulkan want row major
00024     // here
00025     glm::mat4 transpose_transform = glm::transpose(glm::mat4(1.0f));
00026     VkTransformMatrixKHR out_matrix;
00027     std::memcpy(&out_matrix, &transpose_transform, sizeof(VkTransformMatrixKHR));
00028
00029     index_count = static_cast<uint32_t>(indices.size());
00030     vertex_count = static_cast<uint32_t>(vertices.size());
00031     this->device = device;
00032     object_description = ObjectDescription{};
00033     createVertexBuffer(transfer_queue, transfer_command_pool, vertices);
00034     createIndexBuffer(transfer_queue, transfer_command_pool, indices);
00035     createMaterialIDBuffer(transfer_queue, transfer_command_pool, materialIndex);
00036     createMaterialBuffer(transfer_queue, transfer_command_pool, materials);
00037
00038     VkBufferDeviceAddressInfo vertex_info{};
00039     vertex_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00040     vertex_info.buffer = vertexBuffer.getBuffer();
00041
00042     VkBufferDeviceAddressInfo index_info{};
00043     index_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00044     index_info.buffer = indexBuffer.getBuffer();
00045
00046     VkBufferDeviceAddressInfo material_index_info{};
00047     material_index_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00048     material_index_info.buffer = materialIdsBuffer.getBuffer();
00049
00050     VkBufferDeviceAddressInfo material_info{};
00051     material_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00052     material_info.buffer = materialsBuffer.getBuffer();
00053
00054     object_description.index_address =
00055         vkGetBufferDeviceAddress(device->getLogicalDevice(), &index_info);
00056     object_description.vertex_address =
00057         vkGetBufferDeviceAddress(device->getLogicalDevice(), &vertex_info);
00058     object_description.material_index_address = vkGetBufferDeviceAddress(
00059         device->getLogicalDevice(), &material_index_info);
00060     object_description.material_address =
00061         vkGetBufferDeviceAddress(device->getLogicalDevice(), &material_info);
00062
00063     model = glm::mat4(1.0f);
00064 }
```

References [createIndexBuffer\(\)](#), [createMaterialBuffer\(\)](#), [createMaterialIDBuffer\(\)](#), [createVertexBuffer\(\)](#), [device](#), [VulkanBuffer::getBuffer\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [ObjectDescription::index\\_address](#), [index\\_count](#), [indexBuffer](#), [ObjectDescription::material\\_address](#), [ObjectDescription::material\\_index\\_address](#), [materialIdsBuffer](#), [materialsBuffer](#), [model](#), [object\\_description](#), [ObjectDescription::vertex\\_address](#), [vertex\\_count](#), and [vertexBuffer](#).

Here is the call graph for this function:



### 5.15.2.2 Mesh() [2/2]

`Mesh::Mesh ()`

Definition at line 8 of file [Mesh.cpp](#).  
00008 {}

### 5.15.2.3 ~Mesh()

`Mesh::~Mesh ()`

Definition at line 68 of file [Mesh.cpp](#).  
00068 {}

## 5.15.3 Member Function Documentation

### 5.15.3.1 cleanUp()

```
void Mesh::cleanUp ( )
```

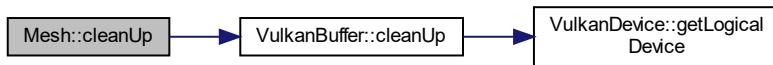
Definition at line 10 of file [Mesh.cpp](#).

```
00010     {
00011     vertexBuffer.cleanUp();
00012     indexBuffer.cleanUp();
00013     objectDescriptionBuffer.cleanUp();
00014     materialIdsBuffer.cleanUp();
00015     materialsBuffer.cleanUp();
00016 }
```

References [VulkanBuffer::cleanUp\(\)](#), [indexBuffer](#), [materialIdsBuffer](#), [materialsBuffer](#), [objectDescriptionBuffer](#), and [vertexBuffer](#).

Referenced by [Model::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.15.3.2 createIndexBuffer()

```
void Mesh::createIndexBuffer (
    VkQueue transfer_queue,
    VkCommandPool transfer_command_pool,
    std::vector< uint32_t > & indices ) [private]
```

Definition at line 84 of file [Mesh.cpp](#).

```
00086
00087     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00088         device, transfer_command_pool, indexBuffer,
00089         VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
00090         VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00091         VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00092         VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00093         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
```

```

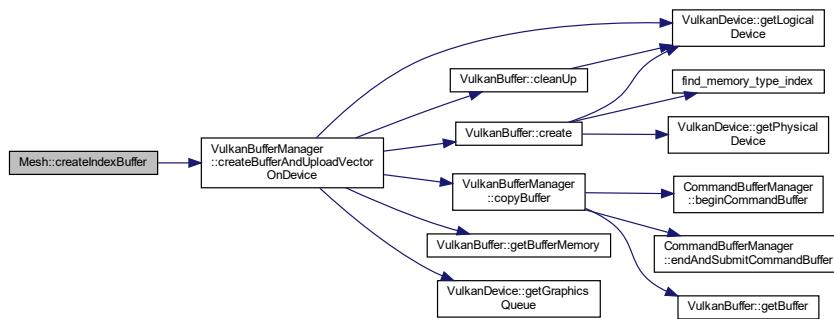
00094     VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00095     indices);
00096 }

```

References [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [device](#), [indexBuffer](#), and [vulkanBufferManager](#).

Referenced by [Mesh\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.15.3.3 `createMaterialBuffer()`

```

void Mesh::createMaterialBuffer (
    VkQueue transfer_queue,
    VkCommandPool transfer_command_pool,
    std::vector< ObjMaterial > & materials ) [private]

```

Definition at line 112 of file [Mesh.cpp](#).

```

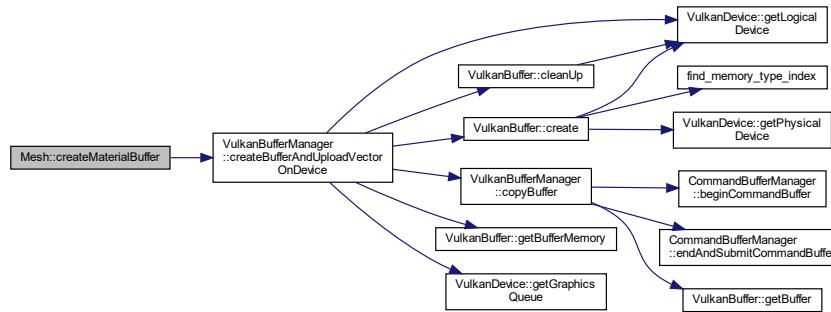
00114     {
00115         vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00116             device, transfer_command_pool, materialsBuffer,
00117             VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
00118             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00119             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00120             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00121             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00122             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00123             materials);
00124     }

```

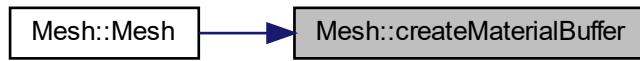
References [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [device](#), [materialIdsBuffer](#), and [vulkanBufferManager](#).

Referenced by [Mesh\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.15.3.4 createMaterialIDBuffer()

```

void Mesh::createMaterialIDBuffer (
    VkQueue transfer_queue,
    VkCommandPool transfer_command_pool,
    std::vector< unsigned int > & materialIndex ) [private]

```

Definition at line 98 of file [Mesh.cpp](#).

```

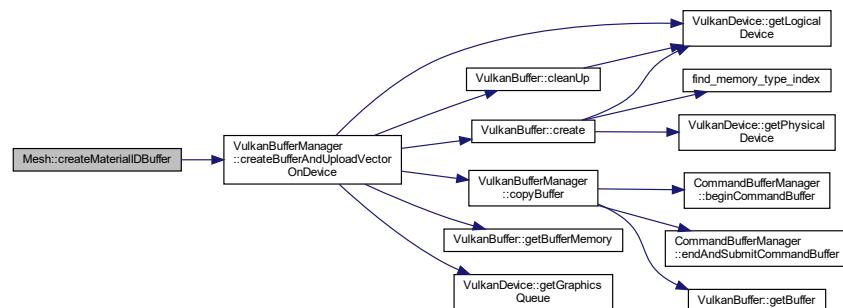
00100
00101     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00102         device, transfer_command_pool, materialIdsBuffer,
00103         VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
00104             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00105             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00106             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00107             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00108             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00109             materialIndex);
00110 }

```

References [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [device](#), [materialIdsBuffer](#), and [vulkanBufferManager](#).

Referenced by [Mesh\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.15.3.5 createVertexBuffer()

```

void Mesh::createVertexBuffer (
    VkQueue transfer_queue,
    VkCommandPool transfer_command_pool,
    std::vector< Vertex > & vertices ) [private]
  
```

Definition at line 70 of file [Mesh.cpp](#).

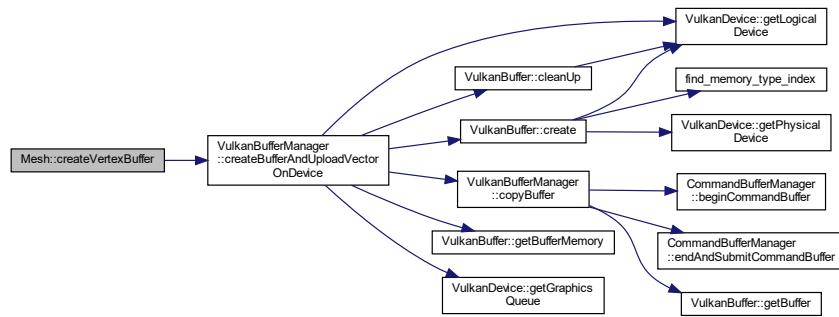
```

00072
00073     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00074         device, transfer_command_pool, vertexBuffer,
00075         VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT |
00076             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00077             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00078             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00079             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00080             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00081             vertices);
00082 }
  
```

References [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [device](#), [vertexBuffer](#), and [vulkanBufferManager](#).

Referenced by [Mesh\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.15.3.6 getIndexBuffer()

```
VkBuffer & Mesh::getIndexBuffer ( ) [inline]
```

Definition at line 29 of file [Mesh.hpp](#).

```
00029 { return indexBuffer.getBuffer(); };
```

References [VulkanBuffer::getBuffer\(\)](#), and [indexBuffer](#).

Referenced by [ASManager::objectToVkGeometryKHR\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.15.3.7 getIndexCount()

`uint32_t Mesh::getIndexCount () [inline]`

Definition at line 26 of file [Mesh.hpp](#).

```
00026 { return index_count; };
```

References [index\\_count](#).

Referenced by [Model::getPrimitiveCount\(\)](#), and [ASManager::objectToVkGeometryKHR\(\)](#).

Here is the caller graph for this function:



### 5.15.3.8 getMaterialIDBuffer()

`VkBuffer & Mesh::getMaterialIDBuffer () [inline]`

Definition at line 28 of file [Mesh.hpp](#).

```
00028 { return materialIdsBuffer.getBuffer(); };
```

References [VulkanBuffer::getBuffer\(\)](#), and [materialIdsBuffer](#).

Here is the call graph for this function:



### 5.15.3.9 getModel()

```
glm::mat4 Mesh::getModel ( ) [inline]
```

Definition at line 24 of file [Mesh.hpp](#).

```
00024 { return model; };
```

References [model](#).

### 5.15.3.10 getObjectDescription()

```
ObjectDescription & Mesh::getObjectDescription ( ) [inline]
```

Definition at line 23 of file [Mesh.hpp](#).

```
00023 { return object_description; };
```

References [object\\_description](#).

Referenced by [Model::getObjectDescription\(\)](#).

Here is the caller graph for this function:



### 5.15.3.11 getVertexBuffer()

```
VkBuffer & Mesh::getVertexBuffer ( ) [inline]
```

Definition at line 27 of file [Mesh.hpp](#).

```
00027 { return vertexBuffer.getBuffer(); };
```

References [VulkanBuffer::getBuffer\(\)](#), and [vertexBuffer](#).

Referenced by [ASManager::objectToVkGeometryKHR\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.15.3.12 getVertexCount()

```
uint32_t Mesh::getVertexCount () [inline]
```

Definition at line 25 of file [Mesh.hpp](#).

```
00025 { return vertex_count; }
```

References [vertex\\_count](#).

Referenced by [ASManager::objectToVkGeometryKHR\(\)](#).

Here is the caller graph for this function:



### 5.15.3.13 setModel()

```
void Mesh::setModel (
    glm::mat4 new_model )
```

Definition at line 66 of file [Mesh.cpp](#).

```
00066 { model = new_model; }
```

References [model](#).

## 5.15.4 Field Documentation

### 5.15.4.1 device

```
VulkanDevice* Mesh::device {VK_NULL_HANDLE} [private]
```

Definition at line 53 of file [Mesh.hpp](#).

Referenced by [createIndexBuffer\(\)](#), [createMaterialBuffer\(\)](#), [createMaterialIDBuffer\(\)](#), [createVertexBuffer\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.2 index\_count

```
uint32_t Mesh::index_count {static_cast<uint32_t>(-1)} [private]
```

Definition at line 51 of file [Mesh.hpp](#).

Referenced by [getIndexCount\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.3 indexBuffer

```
VulkanBuffer Mesh::indexBuffer [private]
```

Definition at line 43 of file [Mesh.hpp](#).

Referenced by [cleanUp\(\)](#), [createIndexBuffer\(\)](#), [getIndexBuffer\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.4 materialIdsBuffer

```
VulkanBuffer Mesh::materialIdsBuffer [private]
```

Definition at line 45 of file [Mesh.hpp](#).

Referenced by [cleanUp\(\)](#), [createMaterialIDBuffer\(\)](#), [getMaterialIDBuffer\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.5 materialsBuffer

```
VulkanBuffer Mesh::materialsBuffer [private]
```

Definition at line 46 of file [Mesh.hpp](#).

Referenced by [cleanUp\(\)](#), [createMaterialBuffer\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.6 model

```
glm::mat4 Mesh::model [private]
```

Definition at line 48 of file [Mesh.hpp](#).

Referenced by [getModel\(\)](#), [Mesh\(\)](#), and [setModel\(\)](#).

#### 5.15.4.7 object\_description

```
ObjectDescription Mesh::object_description [private]
```

##### Initial value:

```
{
    static_cast<uint64_t>(-1), static_cast<uint64_t>(-1),
    static_cast<uint64_t>(-1), static_cast<uint64_t>(-1) }
```

Definition at line 38 of file [Mesh.hpp](#).

Referenced by [getObjectDescription\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.8 objectDescriptionBuffer

```
VulkanBuffer Mesh::objectDescriptionBuffer [private]
```

Definition at line 44 of file [Mesh.hpp](#).

Referenced by [cleanUp\(\)](#).

#### 5.15.4.9 vertex\_count

```
uint32_t Mesh::vertex_count {static_cast<uint32_t>(-1)} [private]
```

Definition at line 50 of file [Mesh.hpp](#).

Referenced by [getVertexCount\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.10 vertexBuffer

```
VulkanBuffer Mesh::vertexBuffer [private]
```

Definition at line 42 of file [Mesh.hpp](#).

Referenced by [cleanUp\(\)](#), [createVertexBuffer\(\)](#), [getVertexBuffer\(\)](#), and [Mesh\(\)](#).

#### 5.15.4.11 vulkanBufferManager

```
VulkanBufferManager Mesh::vulkanBufferManager [private]
```

Definition at line 36 of file [Mesh.hpp](#).

Referenced by [createIndexBuffer\(\)](#), [createMaterialBuffer\(\)](#), [createMaterialIDBuffer\(\)](#), and [createVertexBuffer\(\)](#).

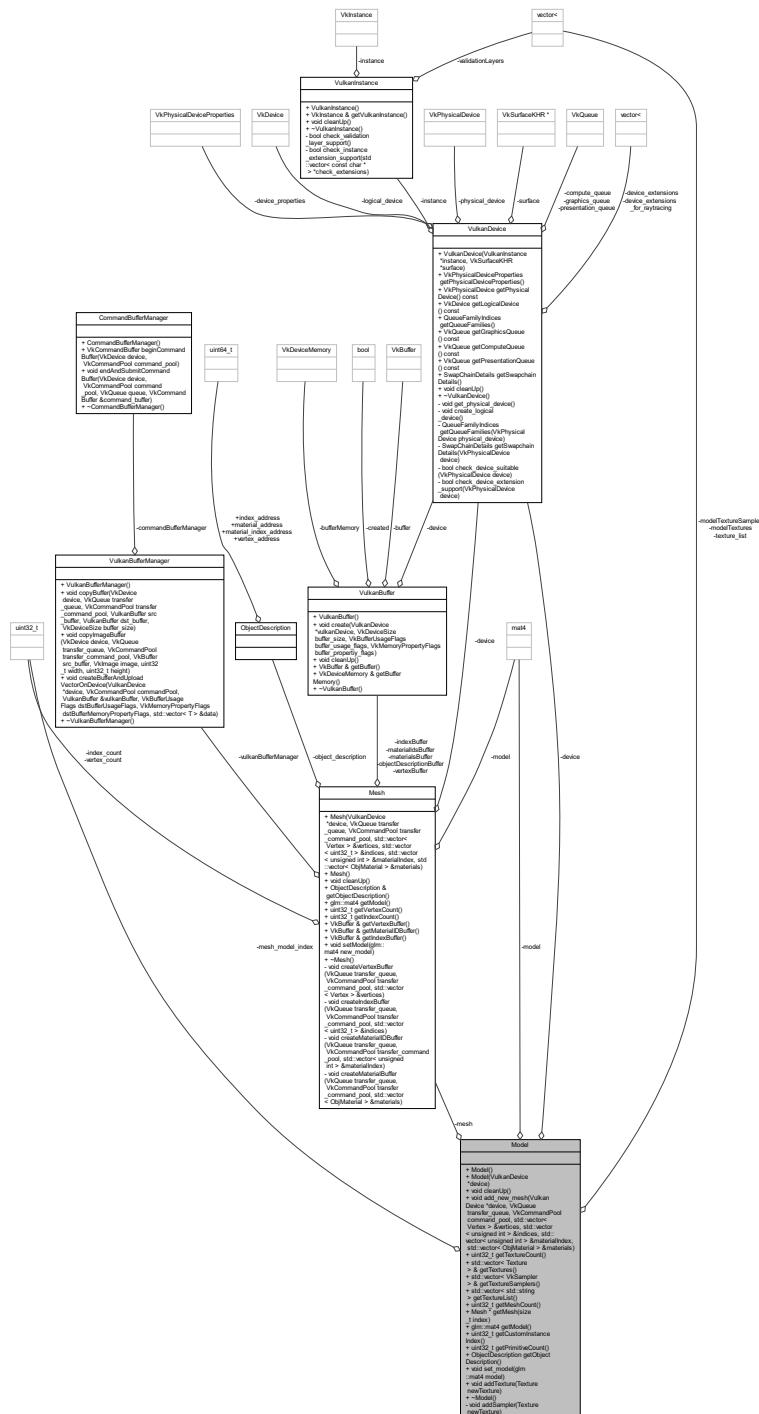
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[Mesh.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/[Mesh.cpp](#)

## 5.16 Model Class Reference

```
#include <Model.hpp>
```

Collaboration diagram for Model:



## Public Member Functions

- [Model \(\)](#)

- `Model (VulkanDevice *device)`
- `void cleanUp ()`
- `void add_new_mesh (VulkanDevice *device, VkQueue transfer_queue, VkCommandPool command_pool, std::vector< Vertex > &vertices, std::vector< unsigned int > &indices, std::vector< unsigned int > &materialIndex, std::vector< ObjMaterial > &materials)`
- `uint32_t getTextureCount ()`
- `std::vector< Texture > &getTextures ()`
- `std::vector< VkSampler > &getTextureSamplers ()`
- `std::vector< std::string > getTextureList ()`
- `uint32_t getMeshCount ()`
- `Mesh * getMesh (size_t index)`
- `glm::mat4 getModel ()`
- `uint32_t getCustomInstanceIndex ()`
- `uint32_t getPrimitiveCount ()`
- `ObjectDescription getObjectDescription ()`
- `void set_model (glm::mat4 model)`
- `void addTexture (Texture newTexture)`
- `~Model ()`

## Private Member Functions

- `void addSampler (Texture newTexture)`

## Private Attributes

- `VulkanDevice * device {VK_NULL_HANDLE}`
- `uint32_t mesh_model_index {static_cast<uint32_t>(-1)}`
- `Mesh mesh`
- `glm::mat4 model`
- `std::vector< std::string > texture_list`
- `std::vector< Texture > modelTextures`
- `std::vector< VkSampler > modelTextureSamplers`

### 5.16.1 Detailed Description

Definition at line 10 of file [Model.hpp](#).

### 5.16.2 Constructor & Destructor Documentation

#### 5.16.2.1 Model() [1/2]

```
Model::Model ( )
```

Definition at line 3 of file [Model.cpp](#).  
00003 { }

### 5.16.2.2 Model() [2/2]

```
Model::Model (   
    VulkanDevice * device )
```

Definition at line 5 of file [Model.cpp](#).  
00005 { this->device = device; }

References [device](#).

### 5.16.2.3 ~Model()

```
Model::~Model ( )
```

Definition at line 49 of file [Model.cpp](#).  
00049 { }

## 5.16.3 Member Function Documentation

### 5.16.3.1 add\_new\_mesh()

```
void Model::add_new_mesh (   
    VulkanDevice * device,  
    VkQueue transfer_queue,  
    VkCommandPool command_pool,  
    std::vector< Vertex > & vertices,  
    std::vector< unsigned int > & indices,  
    std::vector< unsigned int > & materialIndex,  
    std::vector< ObjMaterial > & materials )
```

Definition at line 19 of file [Model.cpp](#).  
00024 {  
00025 this->mesh = Mesh(device, transfer\_queue, command\_pool, vertices, indices,  
00026 materialIndex, materials);  
00027 }

References [device](#), and [mesh](#).

### 5.16.3.2 addSampler()

```
void Model::addSampler (
    Texture newTexture ) [private]
```

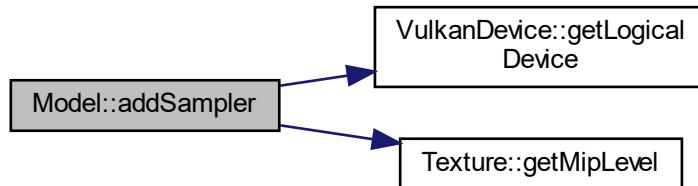
Definition at line 51 of file [Model.cpp](#).

```
00051     {
00052         VkSampler newSampler;
00053         // sampler create info
00054         VkSamplerCreateInfo sampler_create_info{};
00055         sampler_create_info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
00056         sampler_create_info.magFilter = VK_FILTER_LINEAR;
00057         sampler_create_info.minFilter = VK_FILTER_LINEAR;
00058         sampler_create_info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00059         sampler_create_info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00060         sampler_create_info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00061         sampler_create_info.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
00062         sampler_create_info.unnormalizedCoordinates = VK_FALSE;
00063         sampler_create_info.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
00064         sampler_create_info.mipLodBias = 0.0f;
00065         sampler_create_info.minLod = 0.0f;
00066         sampler_create_info.maxLod = newTexture.getMipLevel();
00067         sampler_create_info.anisotropyEnable = VK_TRUE;
00068         sampler_create_info.maxAnisotropy = 16; // max anisotropy sample level
00069         VkResult result = vkCreateSampler(device->getLogicalDevice(),
00070                                         &sampler_create_info, nullptr, &newSampler);
00072         ASSERT_VULKAN(result, "Failed to create a texture sampler!");
00073         modelTextureSamplers.push_back(newSampler);
00075 }
```

References [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [Texture::getMipLevel\(\)](#), and [modelTextureSamplers](#).

Referenced by [addTexture\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.16.3.3 addTexture()

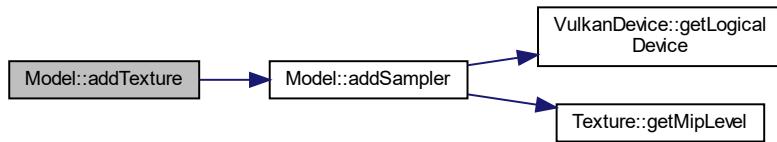
```
void Model::addTexture (
    Texture newTexture )
```

Definition at line 31 of file [Model.cpp](#).

```
00031     {
00032     modelTextures.push_back(newTexture);
00033     addSampler(newTexture);
00034 }
```

References [addSampler\(\)](#), and [modelTextures](#).

Here is the call graph for this function:



### 5.16.3.4 cleanUp()

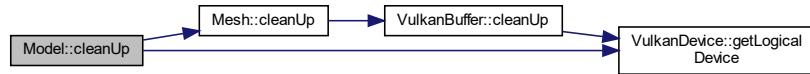
```
void Model::cleanUp ( )
```

Definition at line 7 of file [Model.cpp](#).

```
00007     {
00008     for (Texture texture : modelTextures) {
00009         texture.cleanUp();
00010     }
00011
00012     for (VkSampler texture_sampler : modelTextureSamplers) {
00013         vkDestroySampler(device->getLogicalDevice(), texture_sampler, nullptr);
00014     }
00015
00016     mesh.cleanUp();
00017 }
```

References [Mesh::cleanUp\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [mesh](#), [modelTextures](#), and [modelTextureSamplers](#).

Here is the call graph for this function:



### 5.16.3.5 getCustomInstanceIndex()

```
uint32_t Model::getCustomInstanceIndex ( ) [inline]
```

Definition at line 32 of file [Model.hpp](#).  
00032 { **return** mesh\_model\_index; };

References [mesh\\_model\\_index](#).

### 5.16.3.6 getMesh()

```
Mesh * Model::getMesh ( size_t index ) [inline]
```

Definition at line 30 of file [Model.hpp](#).  
00030 { **return** &mesh; };

References [mesh](#).

### 5.16.3.7 getMeshCount()

```
uint32_t Model::getMeshCount ( ) [inline]
```

Definition at line 29 of file [Model.hpp](#).  
00029 { **return** 1; };

### 5.16.3.8 getModel()

```
glm::mat4 Model::getModel ( ) [inline]
```

Definition at line 31 of file [Model.hpp](#).  
00031 { **return** model; };

References [model](#).

### 5.16.3.9 getObjectDescription()

```
ObjectDescription Model::getObjectDescription () [inline]
```

Definition at line 34 of file [Model.hpp](#).

```
00034     {
00035         return mesh.getObjectDescription();
00036     }
```

References [Mesh::getObjectDescription\(\)](#), and [mesh](#).

Here is the call graph for this function:



### 5.16.3.10 getPrimitiveCount()

```
uint32_t Model::getPrimitiveCount ()
```

Definition at line 36 of file [Model.cpp](#).

```
00036     {
00037     /*uint32_t number_of_indices = 0;
00038
00039     for (Mesh mesh : meshes) {
00040
00041         number_of_indices += mesh.get_index_count();
00042
00043     }
00044
00045     return number_of_indices / 3; */
00046     return mesh.getIndexCount() / 3;
00047 }
```

References [Mesh::getIndexCount\(\)](#), and [mesh](#).

Here is the call graph for this function:



### 5.16.3.11 getTextureCount()

```
uint32_t Model::getTextureCount ( ) [inline]
```

Definition at line 23 of file [Model.hpp](#).

```
00023     {
00024         return static_cast<uint32_t>(modelTextures.size());
00025     };
```

References [modelTextures](#).

### 5.16.3.12 getTextureList()

```
std::vector< std::string > Model::getTextureList ( ) [inline]
```

Definition at line 28 of file [Model.hpp](#).

```
00028 { return texture_list; };
```

References [texture\\_list](#).

### 5.16.3.13 getTextures()

```
std::vector< Texture > & Model::getTextures ( ) [inline]
```

Definition at line 26 of file [Model.hpp](#).

```
00026 { return modelTextures; }
```

References [modelTextures](#).

### 5.16.3.14 getTextureSamplers()

```
std::vector< VkSampler > & Model::getTextureSamplers ( ) [inline]
```

Definition at line 27 of file [Model.hpp](#).

```
00027 { return modelTextureSamplers; }
```

References [modelTextureSamplers](#).

### 5.16.3.15 set\_model()

```
void Model::set_model (
    glm::mat4 model )
```

Definition at line 29 of file [Model.cpp](#).

```
00029 { this->model = model; }
```

References [model](#).

## 5.16.4 Field Documentation

### 5.16.4.1 device

```
VulkanDevice* Model::device {VK_NULL_HANDLE} [private]
```

Definition at line 44 of file [Model.hpp](#).

Referenced by [add\\_new\\_mesh\(\)](#), [addSampler\(\)](#), [cleanUp\(\)](#), and [Model\(\)](#).

### 5.16.4.2 mesh

```
Mesh Model::mesh [private]
```

Definition at line 49 of file [Model.hpp](#).

Referenced by [add\\_new\\_mesh\(\)](#), [cleanUp\(\)](#), [getMesh\(\)](#), [getObjectDescription\(\)](#), and [getPrimitiveCount\(\)](#).

### 5.16.4.3 mesh\_model\_index

```
uint32_t Model::mesh_model_index {static_cast<uint32_t>(-1)} [private]
```

Definition at line 48 of file [Model.hpp](#).

Referenced by [getCustomInstanceIndex\(\)](#).

### 5.16.4.4 model

```
glm::mat4 Model::model [private]
```

Definition at line 50 of file [Model.hpp](#).

Referenced by [getModel\(\)](#), and [set\\_model\(\)](#).

### 5.16.4.5 modelTextures

```
std::vector<Texture> Model::modelTextures [private]
```

Definition at line 53 of file [Model.hpp](#).

Referenced by [addTexture\(\)](#), [cleanUp\(\)](#), [getTextureCount\(\)](#), and [getTextures\(\)](#).

#### 5.16.4.6 modelTextureSamplers

```
std::vector<VkSampler> Model::modelTextureSamplers [private]
```

Definition at line 54 of file [Model.hpp](#).

Referenced by [addSampler\(\)](#), [cleanUp\(\)](#), and [getTextureSamplers\(\)](#).

#### 5.16.4.7 texture\_list

```
std::vector<std::string> Model::texture_list [private]
```

Definition at line 52 of file [Model.hpp](#).

Referenced by [getTextureList\(\)](#).

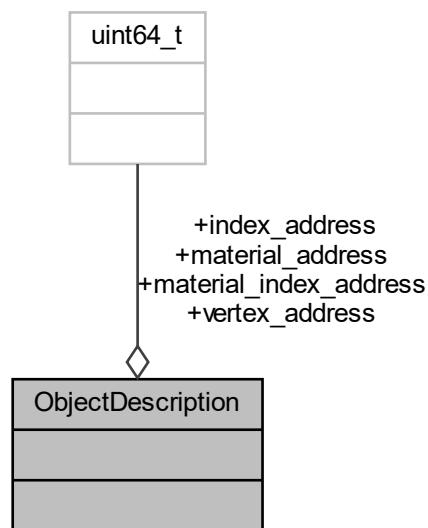
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[Model.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/[Model.cpp](#)

## 5.17 ObjectDescription Struct Reference

```
#include <ObjectDescription.hpp>
```

Collaboration diagram for ObjectDescription:



## Data Fields

- `uint64_t vertex_address`
- `uint64_t index_address`
- `uint64_t material_index_address`
- `uint64_t material_address`

### 5.17.1 Detailed Description

Definition at line 11 of file [ObjectDescription.hpp](#).

### 5.17.2 Field Documentation

#### 5.17.2.1 `index_address`

```
uint64_t ObjectDescription::index_address
```

Definition at line 13 of file [ObjectDescription.hpp](#).

Referenced by [Mesh::Mesh\(\)](#).

#### 5.17.2.2 `material_address`

```
uint64_t ObjectDescription::material_address
```

Definition at line 15 of file [ObjectDescription.hpp](#).

Referenced by [Mesh::Mesh\(\)](#).

#### 5.17.2.3 `material_index_address`

```
uint64_t ObjectDescription::material_index_address
```

Definition at line 14 of file [ObjectDescription.hpp](#).

Referenced by [Mesh::Mesh\(\)](#).

#### 5.17.2.4 vertex\_address

```
uint64_t ObjectDescription::vertex_address
```

Definition at line 12 of file [ObjectDescription.hpp](#).

Referenced by [Mesh::Mesh\(\)](#).

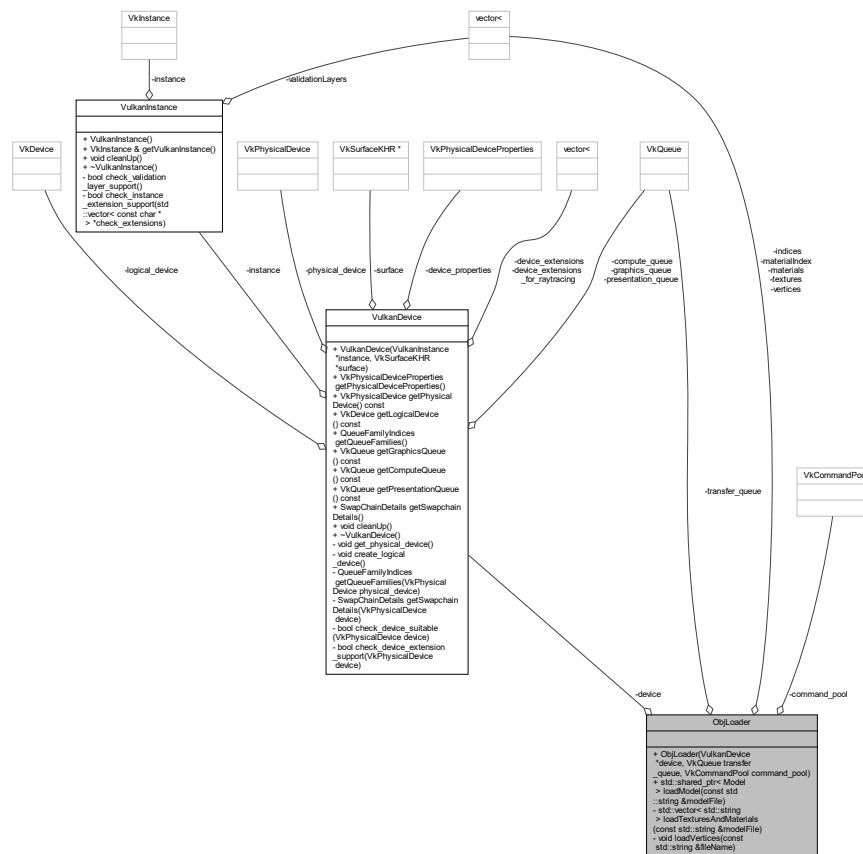
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_16e3q/Desktop/GraphicsEngineVulkan/include/scene/ObjectDescription.hpp

## 5.18 ObjLoader Class Reference

```
#include <ObjLoader.hpp>
```

## Collaboration diagram for ObjLoader:



## Public Member Functions

- `ObjLoader (VulkanDevice *device, VkQueue transfer_queue, VkCommandPool command_pool)`
  - `std::shared_ptr<Model> loadModel (const std::string &modelFile)`

## Private Member Functions

- std::vector< std::string > [loadTexturesAndMaterials](#) (const std::string &modelFile)
- void [loadVertices](#) (const std::string &fileName)

## Private Attributes

- [VulkanDevice](#) \* device
- VkQueue [transfer\\_queue](#)
- VkCommandPool [command\\_pool](#)
- std::vector< [Vertex](#) > [vertices](#)
- std::vector< unsigned int > [indices](#)
- std::vector< [ObjMaterial](#) > [materials](#)
- std::vector< unsigned int > [materialIndex](#)
- std::vector< std::string > [textures](#)

### 5.18.1 Detailed Description

Definition at line 10 of file [ObjLoader.hpp](#).

### 5.18.2 Constructor & Destructor Documentation

#### 5.18.2.1 [ObjLoader\(\)](#)

```
ObjLoader::ObjLoader (
    VulkanDevice * device,
    VkQueue transfer_queue,
    VkCommandPool command_pool )
```

Definition at line 7 of file [ObjLoader.cpp](#).

```
00008
00009     this->device = device;
00010     this->transfer_queue = transfer_queue;
00011     this->command_pool = command_pool;
00012 }
```

References [command\\_pool](#), [device](#), and [transfer\\_queue](#).

### 5.18.3 Member Function Documentation

### 5.18.3.1 loadModel()

```
std::shared_ptr< Model > ObjLoader::loadModel (
    const std::string & modelFile )
```

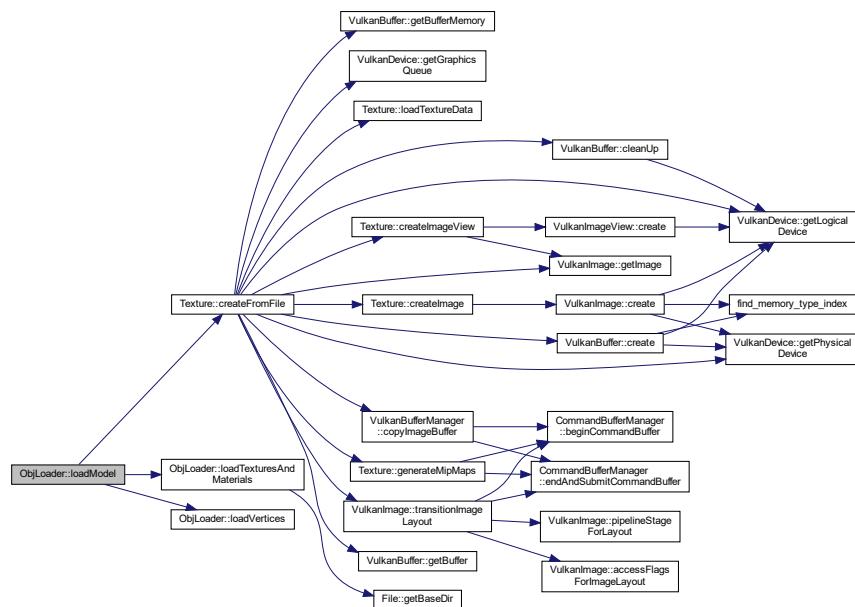
Definition at line 14 of file [ObjLoader.cpp](#).

```
00014
00015 // the model we want to load
00016 std::shared_ptr<Model> new_model = std::make_shared<Model>(device);
00017
00018 // first load textures from model
00019 std::vector<std::string> textureNames = loadTexturesAndMaterials(modelFile);
00020 std::vector<int> matToTex(textureNames.size());
00021
00022 // now that we have the names lets create the vulkan side of textures
00023 for (size_t i = 0; i < textureNames.size(); i++) {
00024     // If material had no texture, set '0' to indicate no texture, texture 0
00025     // will be reserved for a default texture
00026     if (!textureNames[i].empty()) {
00027         // Otherwise, create texture and set value to index of new texture
00028         Texture texture;
00029         texture.createFromFile(device, command_pool, textureNames[i]);
00030         new_model->addTexture(texture);
00031         matToTex[i] = new_model->getTextureCount();
00032
00033     } else {
00034         matToTex[i] = 0;
00035     }
00036 }
00037
00038 loadVertices(modelFile);
00039
00040 new_model->add_new_mesh(device, transfer_queue, command_pool, vertices,
00041                         indices, materialIndex, this->materials);
00042
00043 return new_model;
00044 }
```

References [command\\_pool](#), [Texture::createFromFile\(\)](#), [device](#), [indices](#), [loadTexturesAndMaterials\(\)](#), [loadVertices\(\)](#), [materialIndex](#), [materials](#), [transfer\\_queue](#), and [vertices](#).

Referenced by [Scene::loadModel\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.18.3.2 loadTexturesAndMaterials()

```
std::vector< std::string > ObjLoader::loadTexturesAndMaterials (
    const std::string & modelFile ) [private]
```

Definition at line 46 of file [ObjLoader.cpp](#).

```

00047     {
00048     tinyobj::ObjReaderConfig reader_config;
00049     tinyobj::ObjReader reader;
00050
00051     if (!reader.ParseFromFile(modelFile, reader_config)) {
00052         if (!reader.Error().empty()) {
00053             std::cerr << "TinyObjReader: " << reader.Error();
00054         }
00055         exit(EXIT_FAILURE);
00056     }
00057
00058     if (!reader.Warning().empty()) {
00059         std::cout << "TinyObjReader: " << reader.Warning();
00060     }
00061
00062     auto& tol_materials = reader.GetMaterials();
00063     textures.reserve(tol_materials.size());
00064
00065     int texture_id = 0;
00066
00067     // we now iterate over all materials to get diffuse textures
00068     for (size_t i = 0; i < tol_materials.size(); i++) {
00069         const tinyobj::material_t* mp = &tol_materials[i];
00070         ObjMaterial material{};
00071         material.ambient =
00072             glm::vec3(mp->ambient[0], mp->ambient[1], mp->ambient[2]);
00073         material.diffuse =
00074             glm::vec3(mp->diffuse[0], mp->diffuse[1], mp->diffuse[2]);
00075         material.specular =
00076             glm::vec3(mp->specular[0], mp->specular[1], mp->specular[2]);
00077         material.emission =
00078             glm::vec3(mp->emission[0], mp->emission[1], mp->emission[2]);
00079         material.transmittance = glm::vec3(
00080             mp->transmittance[0], mp->transmittance[1], mp->transmittance[2]);
00081         material.dissolve = mp->dissolve;
00082         material.ior = mp->ior;
00083         material.shininess = mp->shininess;
00084         material.illum = mp->illum;
00085
00086         if (mp->diffuse_texname.length() > 0) {
00087             std::string relative_texture_filename = mp->diffuse_texname;
00088             File model_file(modelFile);
00089             std::string texture_filename =
00090                 model_file.getBaseDir() + "/textures/" + relative_texture_filename;
00091
00092             textures.push_back(texture_filename);
00093             material.textureID = texture_id;
00094             texture_id++;
00095
00096         } else {
00097             material.textureID = 0;
00098             textures.push_back("");
00099         }
00100
00101         materials.push_back(material);
00102     }
00103

```

```

00104 // for the case no .mtl file is given place some random standard material ...
00105 if (tol_materials.empty()) {
00106     materials.emplace_back(ObjMaterial());
00107 }
00108
00109 return textures;
00110 }
```

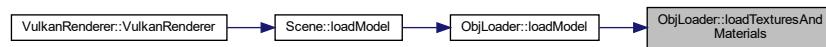
References [ObjMaterial::ambient](#), [File::getBaseDir\(\)](#), [materials](#), and [textures](#).

Referenced by [loadModel\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.18.3.3 loadVertices()

```
void ObjLoader::loadVertices (
    const std::string & fileName ) [private]
```

Definition at line 112 of file [ObjLoader.cpp](#).

```

00112                                     {
00113     tinyobj::ObjReaderConfig reader_config;
00114     // reader_config.mtl_search_path = ""; // Path to material files
00115
00116     tinyobj::ObjReader reader;
00117
00118     if (!reader.ParseFromFile(fileName, reader_config)) {
00119         if (!reader.Error().empty()) {
00120             std::cerr << "TinyObjReader: " << reader.Error();
00121         }
00122         exit(EXIT_FAILURE);
00123     }
00124
00125     if (!reader.Warning().empty()) {
00126         std::cout << "TinyObjReader: " << reader.Warning();
00127     }
00128
00129     auto& attrib = reader.GetAttrib();
00130     auto& shapes = reader.GetShapes();
00131     auto& materials = reader.GetMaterials();
00132
00133     std::unordered_map<Vertex, uint32_t> vertices_map{};


```

```

00134
00135 // Loop over shapes
00136 for (size_t s = 0; s < shapes.size(); s++) {
00137     // prepare for enlargement
00138     vertices.reserve(shapes[s].mesh.indices.size() + vertices.size());
00139     indices.reserve(shapes[s].mesh.indices.size() + indices.size());
00140
00141     // Loop over faces(polygon)
00142     size_t index_offset = 0;
00143     for (size_t f = 0; f < shapes[s].mesh.num_face_vertices.size(); f++) {
00144         size_t fv = size_t(shapes[s].mesh.num_face_vertices[f]);
00145
00146         // Loop over vertices in the face.
00147         for (size_t v = 0; v < fv; v++) {
00148             // access to vertex
00149             tinyobj::index_t idx = shapes[s].mesh.indices[index_offset + v];
00150             tinyobj::real_t vx = attrib.vertices[3 * size_t(idx.vertex_index) + 0];
00151             tinyobj::real_t vy = attrib.vertices[3 * size_t(idx.vertex_index) + 1];
00152             tinyobj::real_t vz = attrib.vertices[3 * size_t(idx.vertex_index) + 2];
00153             glm::vec3 pos = {vx, vy, vz};
00154
00155             glm::vec3 normals(0.0f);
00156             // Check if 'normal_index' is zero or positive. negative = no normal
00157             // data
00158             if (idx.normal_index >= 0 && !attrib.normals.empty()) {
00159                 tinyobj::real_t nx = attrib.normals[3 * size_t(idx.normal_index) + 0];
00160                 tinyobj::real_t ny = attrib.normals[3 * size_t(idx.normal_index) + 1];
00161                 tinyobj::real_t nz = attrib.normals[3 * size_t(idx.normal_index) + 2];
00162                 normals = glm::vec3(nx, ny, nz);
00163             }
00164
00165             glm::vec3 color(-1.f);
00166             if (!attrib.colors.empty()) {
00167                 tinyobj::real_t red = attrib.colors[3 * size_t(idx.vertex_index) + 0];
00168                 tinyobj::real_t green =
00169                     attrib.colors[3 * size_t(idx.vertex_index) + 1];
00170                 tinyobj::real_t blue =
00171                     attrib.colors[3 * size_t(idx.vertex_index) + 2];
00172                 color = glm::vec3(red, green, blue);
00173             }
00174
00175             glm::vec2 tex_coords(0.0f);
00176             // Check if 'texcoord_index' is zero or positive. negative = no texcoord
00177             // data
00178             if (idx.texcoord_index >= 0 && !attrib.texcoords.empty()) {
00179                 tinyobj::real_t tx =
00180                     attrib.texcoords[2 * size_t(idx.texcoord_index) + 0];
00181                 // flip y coordinate !!
00182                 tinyobj::real_t ty =
00183                     1.f - attrib.texcoords[2 * size_t(idx.texcoord_index) + 1];
00184                 tex_coords = glm::vec2(tx, ty);
00185             }
00186
00187             Vertex vert{pos, normals, color, tex_coords};
00188
00189             if (vertices_map.count(vert) == 0) {
00190                 vertices_map[vert] = vertices.size();
00191                 vertices.push_back(vert);
00192             }
00193
00194             indices.push_back(vertices_map[vert]);
00195         }
00196
00197         index_offset += fv;
00198
00199         // per-face material; face usually is triangle
00200         // matToTex[shapes[s].mesh.material_ids[f]]
00201         materialIndex.push_back(shapes[s].mesh.material_ids[f]);
00202     }
00203 }
00204
00205 // precompute normals if no provided
00206 if (attrib.normals.empty()) {
00207     for (size_t i = 0; i < indices.size(); i += 3) {
00208         Vertex& v0 = vertices[indices[i + 0]];
00209         Vertex& v1 = vertices[indices[i + 1]];
00210         Vertex& v2 = vertices[indices[i + 2]];
00211
00212         glm::vec3 n =
00213             glm::normalize(glm::cross((v1.pos - v0.pos), (v2.pos - v0.pos)));
00214         v0.normal = n;
00215         v1.normal = n;
00216         v2.normal = n;
00217     }
00218 }
00219 }
```

References [indices](#), [materialIndex](#), [materials](#), [Vertex::normal](#), [Vertex::pos](#), and [vertices](#).

Referenced by [loadModel\(\)](#).

Here is the caller graph for this function:



## 5.18.4 Field Documentation

### 5.18.4.1 command\_pool

```
VkCommandPool ObjLoader::command_pool [private]
```

Definition at line 20 of file [ObjLoader.hpp](#).

Referenced by [loadModel\(\)](#), and [ObjLoader\(\)](#).

### 5.18.4.2 device

```
VulkanDevice* ObjLoader::device [private]
```

Definition at line 18 of file [ObjLoader.hpp](#).

Referenced by [loadModel\(\)](#), and [ObjLoader\(\)](#).

### 5.18.4.3 indices

```
std::vector<unsigned int> ObjLoader::indices [private]
```

Definition at line 23 of file [ObjLoader.hpp](#).

Referenced by [loadModel\(\)](#), and [loadVertices\(\)](#).

#### 5.18.4.4 materialIndex

```
std::vector<unsigned int> ObjLoader::materialIndex [private]
```

Definition at line 25 of file [ObjLoader.hpp](#).

Referenced by [loadModel\(\)](#), and [loadVertices\(\)](#).

#### 5.18.4.5 materials

```
std::vector<ObjMaterial> ObjLoader::materials [private]
```

Definition at line 24 of file [ObjLoader.hpp](#).

Referenced by [loadModel\(\)](#), [loadTexturesAndMaterials\(\)](#), and [loadVertices\(\)](#).

#### 5.18.4.6 textures

```
std::vector<std::string> ObjLoader::textures [private]
```

Definition at line 26 of file [ObjLoader.hpp](#).

Referenced by [loadTexturesAndMaterials\(\)](#).

#### 5.18.4.7 transfer\_queue

```
VkQueue ObjLoader::transfer_queue [private]
```

Definition at line 19 of file [ObjLoader.hpp](#).

Referenced by [loadModel\(\)](#), and [ObjLoader\(\)](#).

#### 5.18.4.8 vertices

```
std::vector<Vertex> ObjLoader::vertices [private]
```

Definition at line 22 of file [ObjLoader.hpp](#).

Referenced by [loadModel\(\)](#), and [loadVertices\(\)](#).

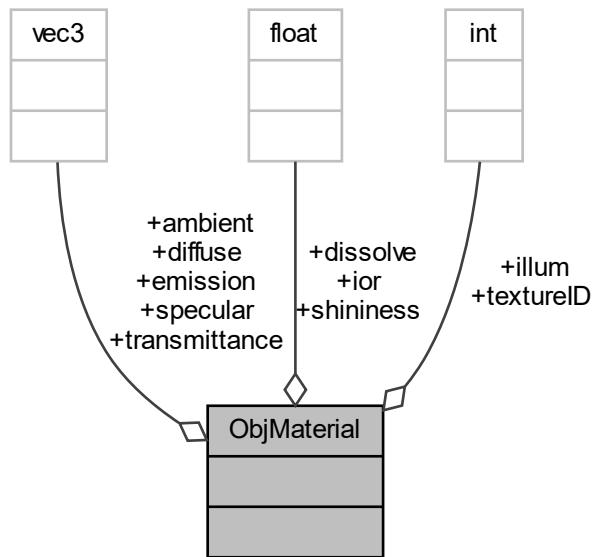
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[ObjLoader.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/[ObjLoader.cpp](#)

## 5.19 ObjMaterial Struct Reference

```
#include <ObjMaterial.hpp>
```

Collaboration diagram for ObjMaterial:



### Data Fields

- `vec3 ambient`
- `vec3 diffuse`
- `vec3 specular`
- `vec3 transmittance`
- `vec3 emission`
- `float shininess`
- `float ior`
- `float dissolve`
- `int illum`
- `int textureID`

### 5.19.1 Detailed Description

Definition at line 19 of file [ObjMaterial.hpp](#).

### 5.19.2 Field Documentation

### 5.19.2.1 ambient

```
vec3 ObjMaterial::ambient
```

Definition at line 20 of file [ObjMaterial.hpp](#).

Referenced by [ObjLoader::loadTexturesAndMaterials\(\)](#).

### 5.19.2.2 diffuse

```
vec3 ObjMaterial::diffuse
```

Definition at line 21 of file [ObjMaterial.hpp](#).

### 5.19.2.3 dissolve

```
float ObjMaterial::dissolve
```

Definition at line 27 of file [ObjMaterial.hpp](#).

### 5.19.2.4 emission

```
vec3 ObjMaterial::emission
```

Definition at line 24 of file [ObjMaterial.hpp](#).

### 5.19.2.5 illum

```
int ObjMaterial::illum
```

Definition at line 29 of file [ObjMaterial.hpp](#).

### 5.19.2.6 ior

```
float ObjMaterial::ior
```

Definition at line 26 of file [ObjMaterial.hpp](#).

### 5.19.2.7 shininess

```
float ObjMaterial::shininess
```

Definition at line 25 of file [ObjMaterial.hpp](#).

### 5.19.2.8 specular

```
vec3 ObjMaterial::specular
```

Definition at line 22 of file [ObjMaterial.hpp](#).

### 5.19.2.9 textureID

```
int ObjMaterial::textureID
```

Definition at line 30 of file ObjMaterial.hpp.

### 5.19.2.10 transmittance

```
vec3 ObjMaterial::transmittance
```

Definition at line 23 of file ObjMaterial.hpp.

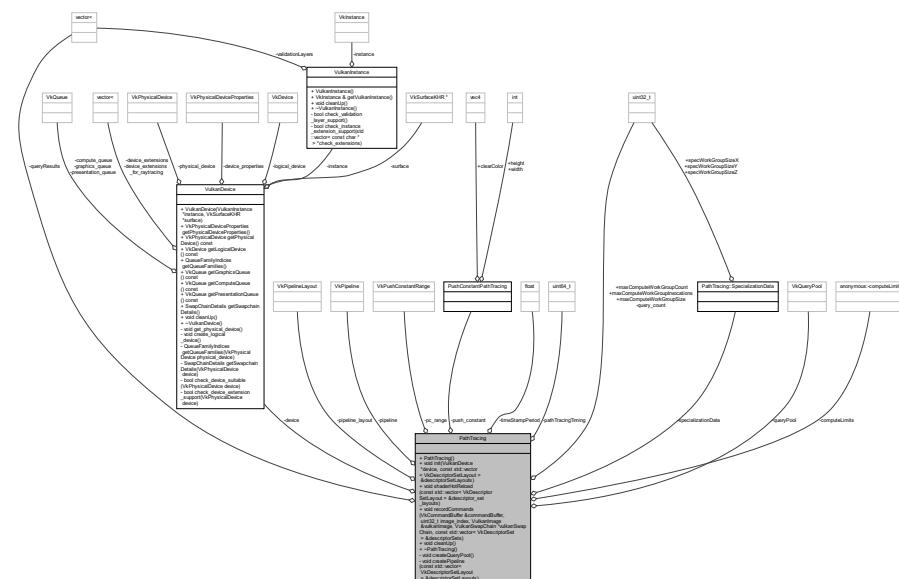
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ObjMaterial.hpp

## 5.20 PathTracing Class Reference

```
#include <PathTracing.hpp>
```

## Collaboration diagram for PathTracing:



## Data Structures

- struct [SpecializationData](#)

## Public Member Functions

- [PathTracing \(\)](#)
- void [init \(VulkanDevice \\*device, const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts\)](#)
- void [shaderHotReload \(const std::vector< VkDescriptorSetLayout > &descriptor\\_set\\_layouts\)](#)
- void [recordCommands \(VkCommandBuffer &commandBuffer, uint32\\_t image\\_index, VulkanImage &vulkanImage, VulkanSwapChain \\*vulkanSwapChain, const std::vector< VkDescriptorSet > &descriptorSets\)](#)
- void [cleanUp \(\)](#)
- [~PathTracing \(\)](#)

## Private Member Functions

- void [createQueryPool \(\)](#)
- void [createPipeline \(const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts\)](#)

## Private Attributes

- [VulkanDevice \\* device {VK\\_NULL\\_HANDLE}](#)
- [VkPipelineLayout pipeline\\_layout {VK\\_NULL\\_HANDLE}](#)
- [VkPipeline pipeline {VK\\_NULL\\_HANDLE}](#)
- [VkPushConstantRange pc\\_range {VK\\_SHADER\\_STAGE\\_FLAG\\_BITS\\_MAX\\_ENUM, 0, 0}](#)
- [PushConstantPathTracing push\\_constant {glm::vec4\(0.f\), 0, 0}](#)
- float [timeStampPeriod {0}](#)
- uint64\_t [pathTracingTiming {static\\_cast<uint64\\_t>\(-1.f\)}](#)
- uint32\_t [query\\_count {2}](#)
- std::vector< uint64\_t > [queryResults](#)
- [VkQueryPool queryPool {VK\\_NULL\\_HANDLE}](#)
- struct {
  - uint32\_t [maxComputeWorkGroupCount \[3\]](#)
  - uint32\_t [maxComputeWorkGroupInvocations = -1](#)
  - uint32\_t [maxComputeWorkGroupSize \[3\]](#)} [computeLimits](#)
- [SpecializationData specializationData](#)

### 5.20.1 Detailed Description

Definition at line 9 of file [PathTracing.hpp](#).

### 5.20.2 Constructor & Destructor Documentation

### 5.20.2.1 PathTracing()

```
PathTracing::PathTracing ( )
```

Definition at line 15 of file [PathTracing.cpp](#).  
00015 {}

### 5.20.2.2 ~PathTracing()

```
PathTracing::~PathTracing ( )
```

Definition at line 169 of file [PathTracing.cpp](#).  
00169 {}

## 5.20.3 Member Function Documentation

### 5.20.3.1 cleanUp()

```
void PathTracing::cleanUp ( )
```

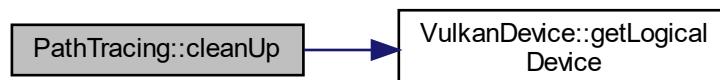
Definition at line 162 of file [PathTracing.cpp](#).

```
00162     {
00163     vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
00164     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00165
00166     vkDestroyQueryPool(device->getLogicalDevice(), queryPool, nullptr);
00167 }
```

References [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [pipeline](#), [pipeline\\_layout](#), and [queryPool](#).

Referenced by [VulkanRenderer::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.20.3.2 createPipeline()

```
void PathTracing::createPipeline (
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts ) [private]
```

Definition at line 185 of file [PathTracing.cpp](#).

```
00186     {
00187         VkPushConstantRange push_constant_range{};
00188         push_constant_range.stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
00189         push_constant_range.offset = 0;
00190         push_constant_range.size = sizeof(PushConstantPathTracing);
00191
00192         VkPipelineLayoutCreateInfo compute_pipeline_layout_create_info{};
00193         compute_pipeline_layout_create_info.sType =
00194             VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00195         compute_pipeline_layout_create_info.setLayoutCount =
00196             static_cast<uint32_t>(descriptorSetLayouts.size());
00197         compute_pipeline_layout_create_info.pushConstantRangeCount = 1;
00198         compute_pipeline_layout_create_info.pPushConstantRanges =
00199             &push_constant_range;
00200         compute_pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00201
00202         ASSERT_VULKAN(vkCreatePipelineLayout(device->getLogicalDevice(),
00203                                             &compute_pipeline_layout_create_info,
00204                                             nullptr, &pipeline_layout),
00205             "Failed to create compute path tracing pipeline layout!");
00206
00207         // create pipeline
00208         std::stringstream pathTracing_shader_dir;
00209         std::filesystem::path cwd = std::filesystem::current_path();
00210         pathTracing_shader_dir << cwd.string();
00211         pathTracing_shader_dir << RELATIVE_RESOURCE_PATH;
00212         pathTracing_shader_dir << "Shaders/path_tracing/";
00213
00214         std::string pathTracing_shader = "path_tracing.comp";
00215
00216         ShaderHelper shaderHelper;
00217         File pathTracingShaderFile(shaderHelper.getShaderSpvDir(
00218             pathTracing_shader_dir.str(), pathTracing_shader));
00219         std::vector<char> pathTracingShadercode =
00220             pathTracingShaderFile.readCharSequence();
00221
00222         shaderHelper.compileShader(pathTracing_shader_dir.str(), pathTracing_shader);
00223
00224         // build shader modules to link to graphics pipeline
00225         VkShaderModule pathTracingModule =
00226             shaderHelper.createShaderModule(device, pathTracingShadercode);
00227
00228         // Specialization constant for workgroup size
00229         std::array<VkSpecializationMapEntry, 2> specEntries{};
00230
00231         specEntries[0].constantID = 0;
00232         specEntries[0].size = sizeof(specializationData.specWorkGroupSizeX);
00233         specEntries[0].offset = 0;
00234
00235         specEntries[1].constantID = 1;
00236         specEntries[1].size = sizeof(specializationData.specWorkGroupSizeY);
00237         specEntries[1].offset = offsetof(SpecializationData, specWorkGroupSizeY);
00238
00239         // specEntries[2].constantID = 2;
00240         // specEntries[2].size = sizeof(specializationData.specWorkGroupSizeZ);
00241         // specEntries[2].offset = offsetof(SpecializationData, specWorkGroupSizeZ);
00242
00243         VkSpecializationInfo specInfo{};
00244         specInfo.dataSize = sizeof(specializationData);
00245         specInfo.mapEntryCount = static_cast<uint32_t>(specEntries.size());
00246         specInfo.pMapEntries = specEntries.data();
00247         specInfo.pData = &specializationData;
00248
00249         VkPipelineShaderStageCreateInfo compute_shader_integrate_create_info{};
00250         compute_shader_integrate_create_info.sType =
00251             VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00252         compute_shader_integrate_create_info.stage = VK_SHADER_STAGE_COMPUTE_BIT;
00253         compute_shader_integrate_create_info.module = pathTracingModule;
00254         compute_shader_integrate_create_info.pSpecializationInfo = &specInfo;
00255         compute_shader_integrate_create_info.pName = "main";
00256
00257         // -- COMPUTE PIPELINE CREATION --
00258         VkComputePipelineCreateInfo compute_pipeline_create_info{};
00259         compute_pipeline_create_info.sType =
00260             VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
00261         compute_pipeline_create_info.stage = compute_shader_integrate_create_info;
00262         compute_pipeline_create_info.layout = pipeline_layout;
00263         compute_pipeline_create_info.flags = 0;
```

```

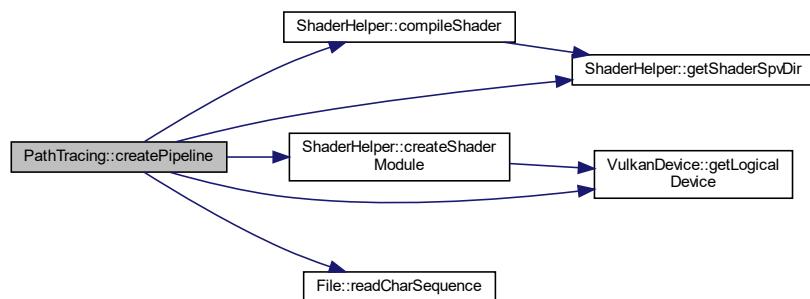
00264 // create compute pipeline
00265 ASSERT_VULKAN(vkCreateComputePipelines(
00266     device->getLogicalDevice(), VK_NULL_HANDLE, 1,
00267     &compute_pipeline_create_info, nullptr, &pipeline),
00268     "Failed to create a compute pipeline!");
00269
00270 // Destroy shader modules, no longer needed after pipeline created
00271 vkDestroyShaderModule(device->getLogicalDevice(), pathTracingModule, nullptr);
00272 }

```

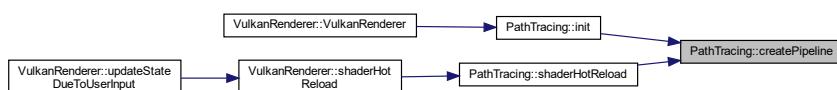
References `ShaderHelper::compileShader()`, `ShaderHelper::createShaderModule()`, `device`, `VulkanDevice::getLogicalDevice()`, `ShaderHelper::getShaderSpvDir()`, `pipeline`, `pipeline_layout`, `File::readCharSequence()`, `specializationData`, `PathTracing::SpecializationData::specWorkGroupSizeX`, and `PathTracing::SpecializationData::specWorkGroupSizeY`.

Referenced by `init()`, and `shaderHotReload()`.

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.20.3.3 `createQueryPool()`

```
void PathTracing::createQueryPool( ) [private]
```

Definition at line 171 of file `PathTracing.cpp`.

```

00171 {
00172     VkQueryPoolCreateInfo queryPoolInfo = {};
00173     queryPoolInfo.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
00174     // This query pool will store pipeline statistics
00175     queryPoolInfo.queryType = VK_QUERY_TYPE_TIMESTAMP;
00176     // Pipeline counters to be returned for this pool
00177     queryPoolInfo.pipelineStatistics =
00178         VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT;
00179     queryPoolInfo.queryCount = query_count;
00180     ASSERT_VULKAN(vkCreateQueryPool(device->getLogicalDevice(), &queryPoolInfo,
00181                                     NULL, &queryPool),
00182                 "Failed to create query pool!");

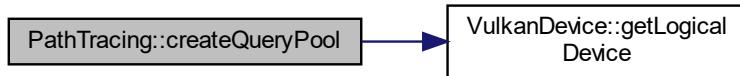
```

```
00183 }
```

References [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [query\\_count](#), and [queryPool](#).

Referenced by [init\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.20.3.4 init()

```
void PathTracing::init (
    VulkanDevice * device,
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts )
```

Definition at line 17 of file [PathTracing.cpp](#).

```

00019     this->device = device;
00020
00021     {
00022         VkPhysicalDeviceProperties physicalDeviceProps =
00023             device->getPhysicalDeviceProperties();
00024         timeStampPeriod = physicalDeviceProps.limits.timestampPeriod;
00025
00026         // save the limits for handling all special cases later on
00027         computeLimits.maxComputeWorkGroupCount[0] =
00028             physicalDeviceProps.limits.maxComputeWorkGroupCount[0];
00029         computeLimits.maxComputeWorkGroupCount[1] =
00030             physicalDeviceProps.limits.maxComputeWorkGroupCount[1];
00031         computeLimits.maxComputeWorkGroupCount[2] =
00032             physicalDeviceProps.limits.maxComputeWorkGroupCount[2];
00033
00034         computeLimits.maxComputeWorkGroupInvocations =
00035             physicalDeviceProps.limits.maxComputeWorkGroupInvocations;
00036
00037         computeLimits.maxComputeWorkGroupSize[0] =
00038             physicalDeviceProps.limits.maxComputeWorkGroupSize[0];
00039         computeLimits.maxComputeWorkGroupSize[1] =
00040             physicalDeviceProps.limits.maxComputeWorkGroupSize[1];
00041         computeLimits.maxComputeWorkGroupSize[2] =
00042             physicalDeviceProps.limits.maxComputeWorkGroupSize[2];
00043
00044         queryResults.resize(query_count);
00045         createQueryPool();

```

```

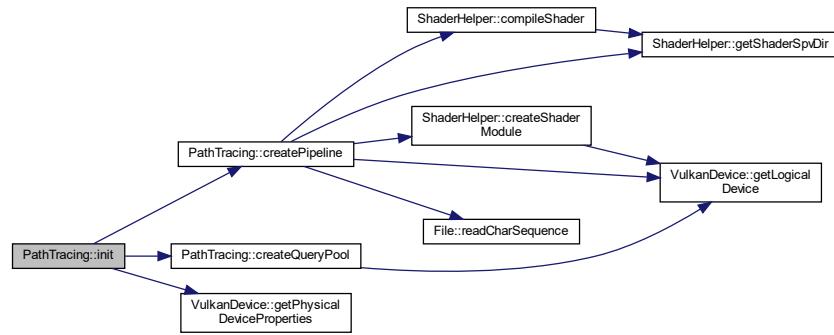
00046
00047     createPipeline(descriptorSetLayouts);
00048 }

```

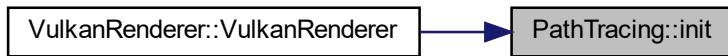
References [computeLimits](#), [createPipeline\(\)](#), [createQueryPool\(\)](#), [device](#), [VulkanDevice::getPhysicalDeviceProperties\(\)](#), [query\\_count](#), [queryResults](#), and [timeStampPeriod](#).

Referenced by [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.20.3.5 recordCommands()

```

void PathTracing::recordCommands (
    VkCommandBuffer & commandBuffer,
    uint32_t image_index,
    VulkanImage & vulkanImage,
    VulkanSwapChain * vulkanSwapChain,
    const std::vector< VkDescriptorSet > & descriptorSets )

```

Definition at line 56 of file [PathTracing.cpp](#).

```

00059
00060     // we have reset the pool; hence start by 0
00061     uint32_t query = 0;
00062
00063     vkCmdResetQueryPool(commandBuffer, queryPool, 0, query_count);
00064
00065     vkCmdWriteTimestamp(
00066         commandBuffer,

```

```

00067     VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,
00068     query++);

00069 QueueFamilyIndices indices = device->getQueueFamilies();

00070 VkImageSubresourceRange subresourceRange{};
00071 subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00072 subresourceRange.baseMipLevel = 0;
00073 subresourceRange.baseArrayLayer = 0;
00074 subresourceRange.levelCount = 1;
00075 subresourceRange.layerCount = 1;

00076 VkImageMemoryBarrier presentToPathTracingImageBarrier{};
00077 presentToPathTracingImageBarrier.sType =
00078     VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00079 presentToPathTracingImageBarrier.pNext = nullptr;
00080 presentToPathTracingImageBarrier.srcQueueFamilyIndex =
00081     indices.graphics_family;
00082 presentToPathTracingImageBarrier.dstQueueFamilyIndex = indices.compute_family;
00083 presentToPathTracingImageBarrier.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
00084 presentToPathTracingImageBarrier.dstAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
00085 presentToPathTracingImageBarrier.oldLayout = VK_IMAGE_LAYOUT_GENERAL;
00086 presentToPathTracingImageBarrier.newLayout = VK_IMAGE_LAYOUT_GENERAL;
00087 presentToPathTracingImageBarrier.subresourceRange = subresourceRange;
00088 presentToPathTracingImageBarrier.image = vulkanImage.getImage();
00089

00090 vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,
00091                         VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
00092                         0, 0, nullptr, 0, nullptr, 1,
00093                         &presentToPathTracingImageBarrier);

00094 VkExtent2D imageSize = vulkanSwapChain->getSwapChainExtent();
00095 push_constant.width = imageSize.width;
00096 push_constant.height = imageSize.height;
00097 push_constant.clearColor = {0.2f, 0.65f, 0.4f, 1.0f};

00098 vkCmdPushConstants(commandBuffer, pipeline_layout,
00099                     VK_SHADER_STAGE_COMPUTE_BIT, 0,
00100                     sizeof(PushConstantPathTracing), &push_constant);

00101 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);
00102

00103 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE,
00104                           pipeline_layout, 0,
00105                           static_cast<uint32_t>(descriptorSets.size()),
00106                           descriptorSets.data(), 0, 0);

00107 uint32_t workGroupCountX =
00108     std::max((imageSize.width + specializationData.specWorkGroupSizeX - 1) /
00109               specializationData.specWorkGroupSizeX,
00110               1U);
00111 uint32_t workGroupCountY =
00112     std::max((imageSize.height + specializationData.specWorkGroupSizeY - 1) /
00113               specializationData.specWorkGroupSizeY,
00114               1U);
00115 uint32_t workGroupCountZ = 1;

00116 vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY,
00117                  workGroupCountZ);

00118 VkImageMemoryBarrier pathTracingToPresentImageBarrier{};
00119 pathTracingToPresentImageBarrier.sType =
00120     VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00121 pathTracingToPresentImageBarrier.pNext = nullptr;
00122 pathTracingToPresentImageBarrier.srcQueueFamilyIndex = indices.compute_family;
00123 pathTracingToPresentImageBarrier.dstQueueFamilyIndex =
00124     indices.graphics_family;
00125 pathTracingToPresentImageBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
00126 pathTracingToPresentImageBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
00127 pathTracingToPresentImageBarrier.oldLayout = VK_IMAGE_LAYOUT_GENERAL;
00128 pathTracingToPresentImageBarrier.newLayout = VK_IMAGE_LAYOUT_GENERAL;
00129 pathTracingToPresentImageBarrier.image = vulkanImage.getImage();
00130 pathTracingToPresentImageBarrier.subresourceRange = subresourceRange;

00131 vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
00132                         VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, 0, 0, nullptr, 0,
00133                         nullptr, 1, &pathTracingToPresentImageBarrier);

00134 vkCmdWriteTimestamp(
00135     commandBuffer,
00136     VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,
00137     query++);

00138 VkResult result = vkGetQueryPoolResults(
00139     device->getLogicalDevice(), queryPool, 0, query_count,
00140     queryResults.size() * sizeof(uint64_t), queryResults.data(),
00141     static_cast<VkDeviceSize>(sizeof(uint64_t)), VK_QUERY_RESULT_64_BIT);
00142

```

```

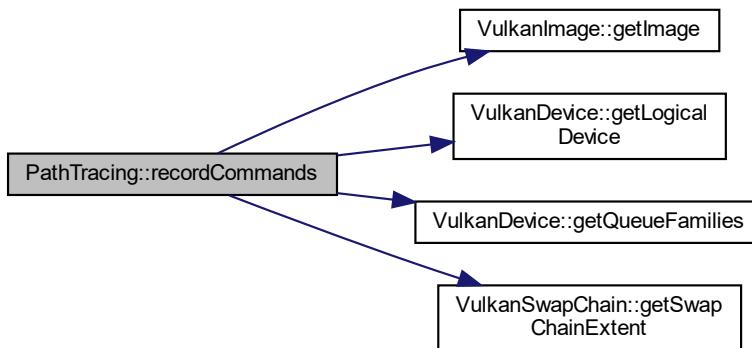
00154
00155     if (result != VK_NOT_READY) {
00156         pathTracingTiming = (static_cast<float>(queryResults[1] - queryResults[0]) *
00157                         timeStampPeriod) /
00158                         1000000.f;
00159     }
00160 }

```

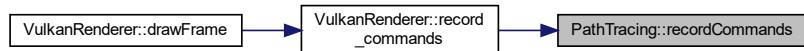
References [PushConstantPathTracing::clearColor](#), [QueueFamilyIndices::compute\\_family](#), [device](#), [VulkanImage::getImage\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanDevice::getQueueFamilies\(\)](#), [VulkanSwapChain::getSwapChainExtent\(\)](#), [QueueFamilyIndices::graphics\\_family](#), [PushConstantPathTracing::height](#), [pathTracingTiming](#), [pipeline](#), [pipeline\\_layout](#), [push\\_constant](#), [query\\_count](#), [queryPool](#), [queryResults](#), [specializationData](#), [PathTracing::SpecializationData::specWorkGroupSizeX](#), [PathTracing::SpecializationData::specWorkGroupSizeY](#), [timeStampPeriod](#), and [PushConstantPathTracing::width](#).

Referenced by [VulkanRenderer::record\\_commands\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.20.3.6 shaderHotReload()

```

void PathTracing::shaderHotReload (
    const std::vector< VkDescriptorSetLayout > & descriptor_set_layouts )

```

Definition at line 50 of file [PathTracing.cpp](#).

```

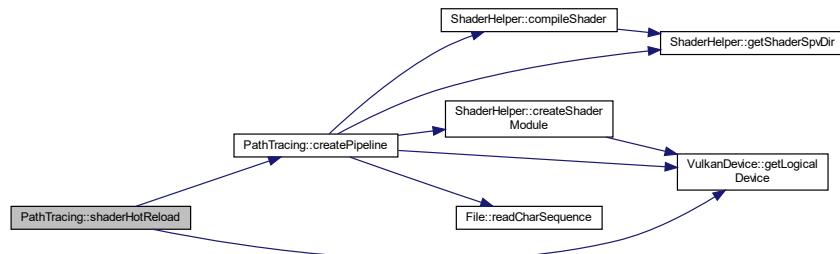
00051
00052     vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
00053     createPipeline(descriptor_set_layouts);
00054 }

```

References [createPipeline\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), and [pipeline](#).

Referenced by [VulkanRenderer::shaderHotReload\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.20.4 Field Documentation

### 5.20.4.1

```
struct { ... } PathTracing::computeLimits [private]
```

Referenced by [init\(\)](#).

### 5.20.4.2 device

```
VulkanDevice* PathTracing::device {VK_NULL_HANDLE} [private]
```

Definition at line 29 of file [PathTracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createPipeline\(\)](#), [createQueryPool\(\)](#), [init\(\)](#), [recordCommands\(\)](#), and [shaderHotReload\(\)](#).

#### 5.20.4.3 maxComputeWorkGroupCount

```
uint32_t PathTracing::maxComputeWorkGroupCount [3]
```

**Initial value:**

```
= {static_cast<uint32_t>(-1),  
     static_cast<uint32_t>(-1),  
     static_cast<uint32_t>(-1)}
```

Definition at line 43 of file [PathTracing.hpp](#).

#### 5.20.4.4 maxComputeWorkGroupInvocations

```
uint32_t PathTracing::maxComputeWorkGroupInvocations = -1
```

Definition at line 46 of file [PathTracing.hpp](#).

#### 5.20.4.5 maxComputeWorkGroupSize

```
uint32_t PathTracing::maxComputeWorkGroupSize[3]
```

**Initial value:**

```
= {static_cast<uint32_t>(-1),  
     static_cast<uint32_t>(-1),  
     static_cast<uint32_t>(-1)}
```

Definition at line 47 of file [PathTracing.hpp](#).

#### 5.20.4.6 pathTracingTiming

```
uint64_t PathTracing::pathTracingTiming {static_cast<uint64_t>(-1.f)} [private]
```

Definition at line 37 of file [PathTracing.hpp](#).

Referenced by [recordCommands\(\)](#).

#### 5.20.4.7 pc\_range

```
VkPushConstantRange PathTracing::pc_range {VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0, 0} [private]
```

Definition at line 33 of file [PathTracing.hpp](#).

#### 5.20.4.8 pipeline

```
VkPipeline PathTracing::pipeline {VK_NULL_HANDLE} [private]
```

Definition at line 32 of file [PathTracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createPipeline\(\)](#), [recordCommands\(\)](#), and [shaderHotReload\(\)](#).

#### 5.20.4.9 pipeline\_layout

```
VkPipelineLayout PathTracing::pipeline_layout {VK_NULL_HANDLE} [private]
```

Definition at line 31 of file [PathTracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createPipeline\(\)](#), and [recordCommands\(\)](#).

#### 5.20.4.10 push\_constant

```
PushConstantPathTracing PathTracing::push_constant {glm::vec4(0.f), 0, 0} [private]
```

Definition at line 34 of file [PathTracing.hpp](#).

Referenced by [recordCommands\(\)](#).

#### 5.20.4.11 query\_count

```
uint32_t PathTracing::query_count {2} [private]
```

Definition at line 38 of file [PathTracing.hpp](#).

Referenced by [createQueryPool\(\)](#), [init\(\)](#), and [recordCommands\(\)](#).

#### 5.20.4.12 queryPool

```
VkQueryPool PathTracing::queryPool {VK_NULL_HANDLE} [private]
```

Definition at line 40 of file [PathTracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createQueryPool\(\)](#), and [recordCommands\(\)](#).

#### 5.20.4.13 queryResults

```
std::vector<uint64_t> PathTracing::queryResults [private]
```

Definition at line 39 of file [PathTracing.hpp](#).

Referenced by [init\(\)](#), and [recordCommands\(\)](#).

#### 5.20.4.14 specializationData

```
SpecializationData PathTracing::specializationData [private]
```

Definition at line 60 of file [PathTracing.hpp](#).

Referenced by [createPipeline\(\)](#), and [recordCommands\(\)](#).

#### 5.20.4.15 timeStampPeriod

```
float PathTracing::timeStampPeriod {0} [private]
```

Definition at line 36 of file [PathTracing.hpp](#).

Referenced by [init\(\)](#), and [recordCommands\(\)](#).

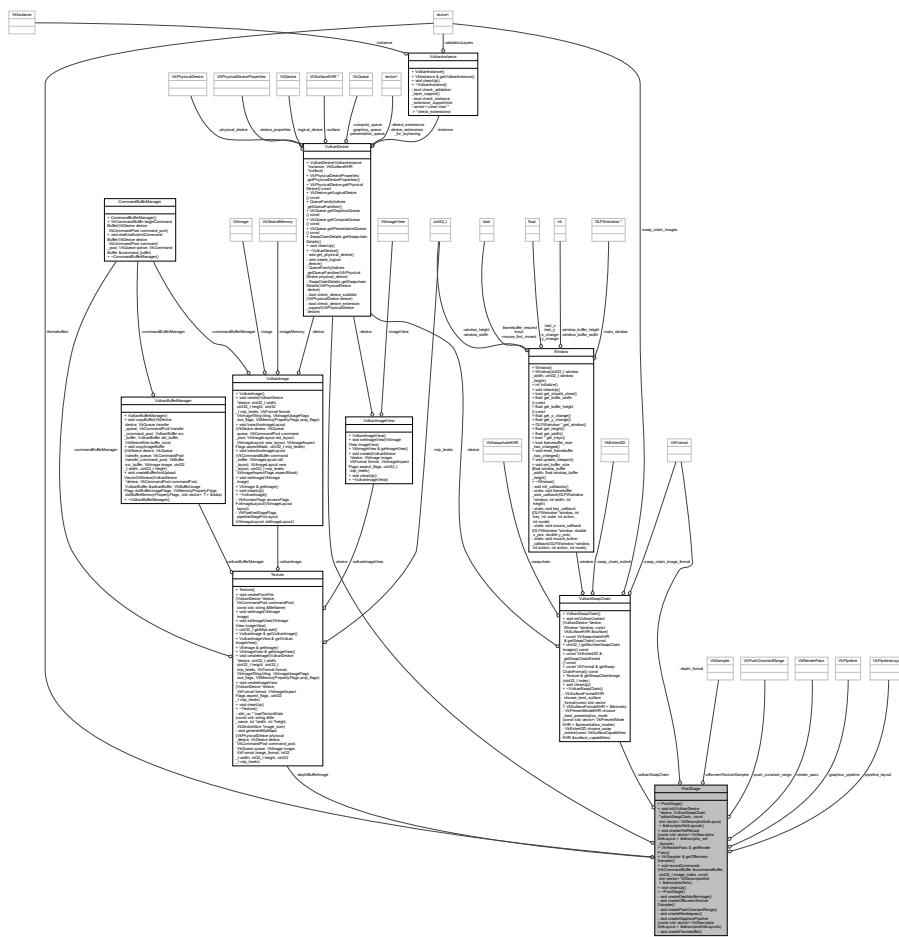
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[PathTracing.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/[PathTracing.cpp](#)

## 5.21 PostStage Class Reference

```
#include <PostStage.hpp>
```

## Collaboration diagram for PostStage:



## Public Member Functions

- `PostStage ()`
  - `void init (VulkanDevice *device, VulkanSwapChain *vulkanSwapChain, const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts)`
  - `void shaderHotReload (const std::vector< VkDescriptorSetLayout > &descriptor_set_layouts)`
  - `VkRenderPass & getRenderPass ()`
  - `VkSampler & getOffscreenSampler ()`
  - `void recordCommands (VkCommandBuffer &commandBuffer, uint32_t image_index, const std::vector< VkDescriptorSet > &descriptorSets)`
  - `void cleanUp ()`
  - `~PostStage ()`

## Private Member Functions

- void `createDepthbufferImage ()`
  - void `createOffscreenTextureSampler ()`
  - void `createPushConstantRange ()`
  - void `createRenderpass ()`
  - void `createGraphicsPipeline (const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts)`
  - void `createFramebuffer ()`

## Private Attributes

- `VulkanDevice * device {VK_NULL_HANDLE}`
- `VulkanSwapChain * vulkanSwapChain {VK_NULL_HANDLE}`
- `std::vector< VkFramebuffer > framebuffers`
- `Texture depthBufferImage`
- `VkFormat depth_format {VK_FORMAT_UNDEFINED}`
- `VkSampler offscreenTextureSampler`
- `VkPushConstantRange push_constant_range`
- `VkRenderPass render_pass {VK_NULL_HANDLE}`
- `VkPipeline graphics_pipeline {VK_NULL_HANDLE}`
- `VkPipelineLayout pipeline_layout {VK_NULL_HANDLE}`

### 5.21.1 Detailed Description

Definition at line 8 of file [PostStage.hpp](#).

### 5.21.2 Constructor & Destructor Documentation

#### 5.21.2.1 PostStage()

```
PostStage::PostStage ( )
```

Definition at line 15 of file [PostStage.cpp](#).  
00015 { }

#### 5.21.2.2 ~PostStage()

```
PostStage::~PostStage ( )
```

Definition at line 107 of file [PostStage.cpp](#).  
00107 { }

### 5.21.3 Member Function Documentation

### 5.21.3.1 cleanUp()

```
void PostStage::cleanUp( )
```

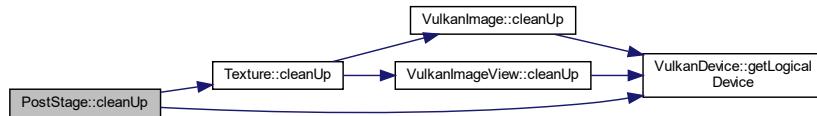
Definition at line 93 of file [PostStage.cpp](#).

```
00093     {
00094     depthBufferImage.cleanUp();
00095     for (auto framebuffer : framebuffers) {
00096         vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
00097     }
00098
00099     vkDestroySampler(device->getLogicalDevice(), offscreenTextureSampler,
00100             nullptr);
00101
00102     vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
00103     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00104     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00105 }
```

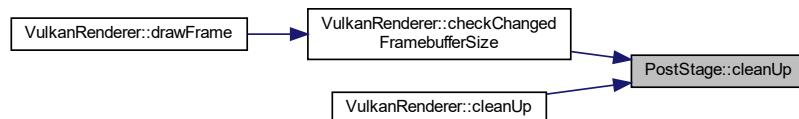
References [Texture::cleanUp\(\)](#), [depthBufferImage](#), [device](#), [framebuffers](#), [VulkanDevice::getLogicalDevice\(\)](#), [graphics\\_pipeline](#), [offscreenTextureSampler](#), [pipeline\\_layout](#), and [render\\_pass](#).

Referenced by [VulkanRenderer::checkChangedFramebufferSize\(\)](#), and [VulkanRenderer::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.2 createDepthbufferImage()

```
void PostStage::createDepthbufferImage( ) [private]
```

Definition at line 109 of file [PostStage.cpp](#).

```
00109     {
00110     // get supported format for depth buffer
00111     depth_format = choose_supported_format(
00112         device->getPhysicalDevice(),
00113         {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00114          VK_FORMAT_D24_UNORM_S8_UINT},
00115         VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
```

```

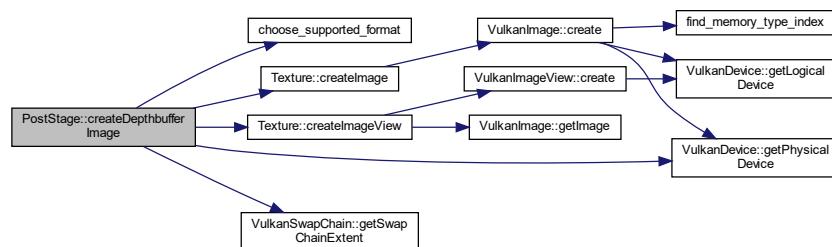
00116
00117 // create depth buffer image
00118 // MIP LEVELS: for depth texture we only want 1 level :(
00119 const VKExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00120 depthBufferImage.createImage(device, swap_chain_extent.width,
00121                               swap_chain_extent.height, 1, depth_format,
00122                               VK_IMAGE_TILING_OPTIMAL,
00123                               VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
00124                               VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00125
00126 // depth buffer image view
00127 // MIP LEVELS: for depth texture we only want 1 level :(
00128 depthBufferImage.createImageView(device, depth_format,
00129                                   VK_IMAGE_ASPECT_DEPTH_BIT, 1);
00130 }

```

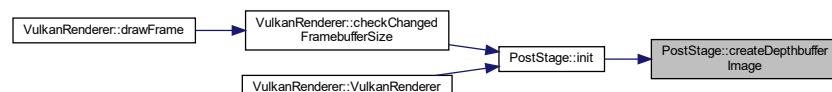
References `choose_supported_format()`, `Texture::createImage()`, `Texture::createImageView()`, `depth_format`, `depthBufferImage`, `device`, `VulkanDevice::getPhysicalDevice()`, `VulkanSwapChain::getSwapChainExtent()`, and `vulkanSwapChain`.

Referenced by `init()`.

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.3 createFramebuffer()

```
void PostStage::createFramebuffer ( ) [private]
```

Definition at line 482 of file `PostStage.cpp`.

```

00482
00483 // resize framebuffer size to equal swap chain image count
00484 framebuffers.resize(vulkanSwapChain->getNumberSwapChainImages());
00485
00486 for (size_t i = 0; i < vulkanSwapChain->getNumberSwapChainImages(); i++) {
00487     Texture& swap_chain_image = vulkanSwapChain->getSwapChainImage(i);
00488     std::array<VkImageView, 2> attachments = {swap_chain_image.getImageView(),

```

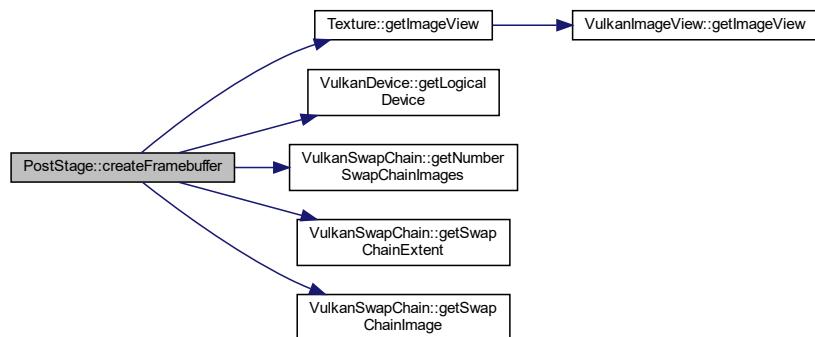
```

00490                               depthBufferImage.getImageView());
00491
00492     VkFramebufferCreateInfo frame_buffer_create_info{};
00493     frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
00494     frame_buffer_create_info.renderPass =
00495         render_pass; // render pass layout the framebuffer will be used with
00496     frame_buffer_create_info.attachmentCount =
00497         static_cast<uint32_t>(attachments.size());
00498     frame_buffer_create_info.pAttachments =
00499         attachments.data(); // list of attachments (1:1 with render pass)
00500     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00501     frame_buffer_create_info.width =
00502         swap_chain_extent.width; // framebuffer width
00503     frame_buffer_create_info.height =
00504         swap_chain_extent.height; // framebuffer height
00505     frame_buffer_create_info.layers = 1; // framebuffer layer
00506
00507     VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
00508                                             &frame_buffer_create_info, nullptr,
00509                                             &framebuffers[i]);
00510     ASSERT_VULKAN(result, "Failed to create framebuffer!")
00511 }
00512 }
```

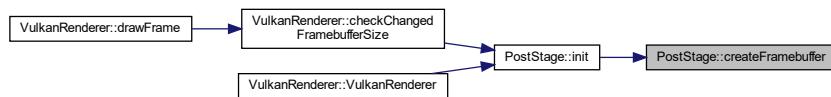
References [depthBufferImage](#), [device](#), [framebuffers](#), [Texture::getImageView\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanSwapChain::getNumberSwapChainImages\(\)](#), [VulkanSwapChain::getSwapChainExtent\(\)](#), [VulkanSwapChain::getSwapChainImage](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [init\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.4 createGraphicsPipeline()

```
void PostStage::createGraphicsPipeline (
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts ) [private]
```

Definition at line 270 of file [PostStage.cpp](#).

```
00271     std::string post_shader_dir;
00272     std::filesystem::path cwd = std::filesystem::current_path();
00273     post_shader_dir += cwd.string();
00274     post_shader_dir += RELATIVE_RESOURCE_PATH;
00275     post_shader_dir += "Shaders/post/";
00276
00277     std::string post_vert_shader = "post.vert";
00278     std::string post_frag_shader = "post.frag";
00279
00280     ShaderHelper shaderHelper;
00281     File vertexShaderFile(
00282         shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_vert_shader));
00283     std::vector<char> vertex_shader_code = vertexShaderFile.readCharSequence();
00284     File fragmentShaderFile(
00285         shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_frag_shader));
00286     std::vector<char> fragment_shader_code =
00287         fragmentShaderFile.readCharSequence();
00288
00289     shaderHelper.compileShader(post_shader_dir.str(), post_vert_shader);
00290     shaderHelper.compileShader(post_shader_dir.str(), post_frag_shader);
00291
00292 // build shader modules to link to graphics pipeline
00293     VkShaderModule vertex_shader_module =
00294         shaderHelper.createShaderModule(device, vertex_shader_code);
00295     VkShaderModule fragment_shader_module =
00296         shaderHelper.createShaderModule(device, fragment_shader_code);
00297
00298 // shader stage creation information
00299 // vertex stage creation information
00300     VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
00301     vertex_shader_create_info.sType =
00302         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00303     vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
00304     vertex_shader_create_info.module = vertex_shader_module;
00305     vertex_shader_create_info.pName = "main";
00306
00307 // fragment stage creation information
00308     VkPipelineShaderStageCreateInfo fragment_shader_create_info{};
00309     fragment_shader_create_info.sType =
00310         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00311     fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
00312     fragment_shader_create_info.module = fragment_shader_module;
00313     fragment_shader_create_info.pName = "main";
00314
00315     std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
00316         vertex_shader_create_info, fragment_shader_create_info};
00317
00318 // how the data for a single vertex (including info such as position, color,
00319 // texture coords, normals, etc) is as a whole
00320     VkVertexInputBindingDescription binding_description{};
00321     binding_description.binding = 0;
00322     binding_description.stride = sizeof(Vertex);
00323     binding_description.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
00324
00325     std::array<VkVertexInputAttributeDescription, 4> attribute_descriptions =
00326         vertex::getVertexInputAttributeDesc();
00327
00328 // CREATE PIPELINE
00329 // 1.) Vertex input
00330     VkPipelineVertexInputStateCreateInfo vertex_input_create_info{};
00331     vertex_input_create_info.sType =
00332         VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
00333     vertex_input_create_info.vertexBindingDescriptionCount = 0;
00334     vertex_input_create_info.pVertexBindingDescriptions = nullptr;
00335     vertex_input_create_info.vertexAttributeDescriptionCount = 0;
00336     vertex_input_create_info.pVertexAttributeDescriptions = nullptr;
00337
00338 // input assembly
00339     VkPipelineInputAssemblyStateCreateInfo input_assembly{};
00340     input_assembly.sType =
00341         VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
00342     input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
00343     input_assembly.primitiveRestartEnable = VK_FALSE;
00344
00345 // viewport & scissor
00346 // create a viewport info struct
00347     VkViewport viewport{};
```

```

00349     viewport.x = 0.0f;
00350     viewport.y = 0.0f;
00351     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00352     viewport.width = (float)swap_chain_extent.width;
00353     viewport.height = (float)swap_chain_extent.height;
00354     viewport.minDepth = 0.0f;
00355     viewport.maxDepth = 1.0f;
00356
00357     // create a scissor info struct
00358     VkRect2D scissor{};
00359     scissor.offset = {0, 0};
00360     scissor.extent = swap_chain_extent;
00361
00362     VkPipelineViewportStateCreateInfo viewport_state_create_info{};
00363     viewport_state_create_info.sType =
00364         VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
00365     viewport_state_create_info.viewportCount = 1;
00366     viewport_state_create_info.pViewports = &viewport;
00367     viewport_state_create_info.scissorCount = 1;
00368     viewport_state_create_info.pScissors = &scissor;
00369
00370     // RASTERIZER
00371     VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
00372     rasterizer_create_info.sType =
00373         VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
00374     rasterizer_create_info.depthClampEnable = VK_FALSE;
00375     rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
00376     rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
00377     rasterizer_create_info.lineWidth = 1.0f;
00378     rasterizer_create_info.cullMode = VK_CULL_MODE_NONE;
00379     // winding to determine which side is front; y-coordinate is inverted in
00380     // comparison to OpenGL
00381     rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
00382     rasterizer_create_info.depthBiasClamp = VK_FALSE;
00383
00384     // -- MULTISAMPLING --
00385     VkPipelineMultisampleStateCreateInfo multisample_create_info{};
00386     multisample_create_info.sType =
00387         VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
00388     multisample_create_info.sampleShadingEnable = VK_FALSE;
00389     multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
00390
00391     // -- BLENDING --
00392     // blend attachment state
00393     VkPipelineColorBlendAttachmentState color_state{};
00394     color_state.colorWriteMask =
00395         VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
00396         VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
00397
00398     color_state.blendEnable = VK_TRUE;
00399     // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
00400     // (dstColorBlendFactor * old_color)
00401     color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
00402     color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
00403     color_state.colorBlendOp = VK_BLEND_OP_ADD;
00404     color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
00405     color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
00406     color_state.alphaBlendOp = VK_BLEND_OP_ADD;
00407
00408     VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
00409     color_blending_create_info.sType =
00410         VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
00411     color_blending_create_info.logicOpEnable =
00412         VK_FALSE; // alternative to calculations is to use logical operations
00413     color_blending_create_info.logicOp = VK_LOGIC_OP_CLEAR;
00414     color_blending_create_info.attachmentCount = 1;
00415     color_blending_create_info.pAttachments = &color_state;
00416     for (int i = 0; i < 4; i++) {
00417         color_blending_create_info.blendConstants[0] = 0.f;
00418     }
00419     // -- PIPELINE LAYOUT --
00420     VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00421     pipeline_layout_create_info.sType =
00422         VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00423     pipeline_layout_create_info.setLayoutCount =
00424         static_cast<uint32_t>(descriptorSetLayouts.size());
00425     pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00426     pipeline_layout_create_info.pushConstantRangeCount = 1;
00427     pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;
00428
00429     // create pipeline layout
00430     VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00431                                             &pipeline_layout_create_info,
00432                                             nullptr, &pipeline_layout);
00433     ASSERT_VULKAN(result, "Failed to create pipeline layout!")
00434
00435     // -- DEPTH STENCIL TESTING --

```

```

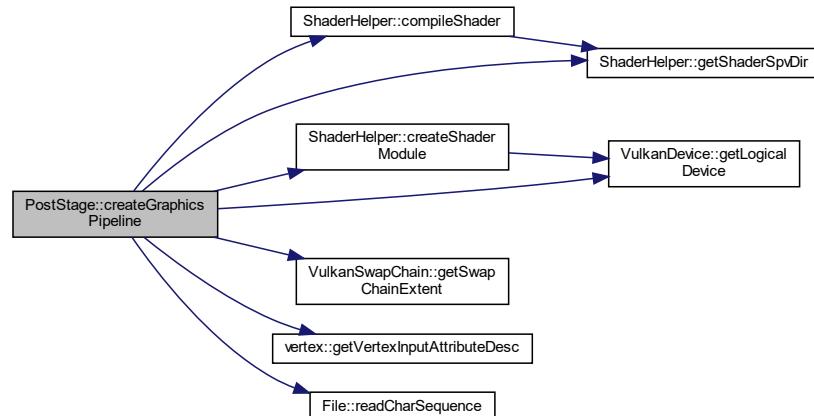
00436 VkPipelineDepthStencilStateCreateInfo depth_stencil_create_info{};
00437 depth_stencil_create_info.sType =
00438     VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
00439 depth_stencil_create_info.depthTestEnable = VK_TRUE;
00440 depth_stencil_create_info.depthWriteEnable = VK_TRUE;
00441 depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
00442 depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
00443 depth_stencil_create_info.stencilTestEnable = VK_FALSE;
00444
00445 // -- GRAPHICS PIPELINE CREATION --
00446 VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
00447 graphics_pipeline_create_info.sType =
00448     VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
00449 graphics_pipeline_create_info.stageCount =
00450     static_cast<uint32_t>(shader_stages.size());
00451 graphics_pipeline_create_info.pStages = shader_stages.data();
00452 graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
00453 graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
00454 graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
00455 graphics_pipeline_create_info.pDynamicState = nullptr;
00456 graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
00457 graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
00458 graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
00459 graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
00460 graphics_pipeline_create_info.layout = pipeline_layout;
00461 graphics_pipeline_create_info.renderPass = render_pass;
00462 graphics_pipeline_create_info.subpass = 0;
00463
00464 // pipeline derivatives : can create multiple pipelines that derive from one
00465 // another for optimization
00466 graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
00467 graphics_pipeline_create_info.basePipelineIndex = -1;
00468
00469 // create graphics pipeline
00470 result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
00471                                     1, &graphics_pipeline_create_info, nullptr,
00472                                     &graphics_pipeline);
00473 ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")
00474
00475 // Destroy shader modules, no longer needed after pipeline created
00476 vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
00477                         nullptr);
00478 vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,
00479                         nullptr);
00480 }

```

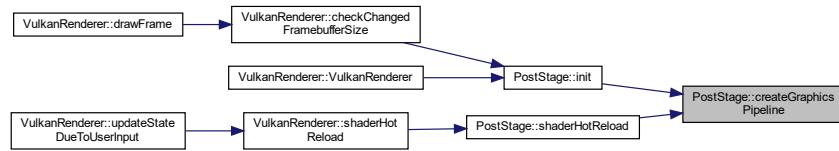
References [ShaderHelper::compileShader\(\)](#), [ShaderHelper::createShaderModule\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [ShaderHelper::getShaderSpvDir\(\)](#), [VulkanSwapChain::getSwapChainExtent\(\)](#), [vertex::getVertexInputAttributeDesc\(\)](#), [graphics\\_pipeline](#), [pipeline\\_layout](#), [push\\_constant\\_range](#), [File::readCharSequence\(\)](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [init\(\)](#), and [shaderHotReload\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.5 `createOffscreenTextureSampler()`

```
void PostStage::createOffscreenTextureSampler() [private]
```

Definition at line 132 of file [PostStage.cpp](#).

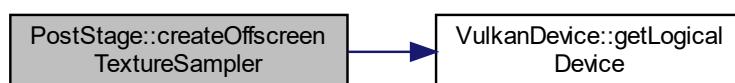
```

00132     {
00133     // sampler create info
00134     VkSamplerCreateInfo sampler_create_info{};
00135     sampler_create_info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
00136     sampler_create_info.magFilter = VK_FILTER_LINEAR;
00137     sampler_create_info.minFilter = VK_FILTER_LINEAR;
00138     sampler_create_info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00139     sampler_create_info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00140     sampler_create_info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00141     sampler_create_info.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
00142     sampler_create_info.unnormalizedCoordinates = VK_FALSE;
00143     sampler_create_info.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
00144     sampler_create_info.mipLodBias = 0.0f;
00145     sampler_create_info.minLod = 0.0f;
00146     sampler_create_info.maxLod = 0.0f;
00147     sampler_create_info.anisotropyEnable = VK_TRUE;
00148     sampler_create_info.maxAnisotropy = 16; // max anisotropy sample level
00149
00150     VkResult result =
00151         vkCreateSampler(device->getLogicalDevice(), &sampler_create_info, nullptr,
00152                         &offscreenTextureSampler);
00153     ASSERT_VULKAN(result, "Failed to create a texture sampler!")
00154 }
```

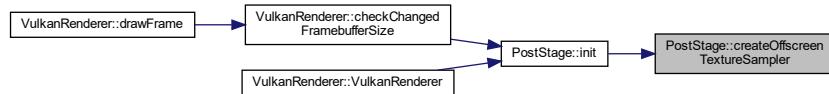
References `device`, [VulkanDevice::getLogicalDevice\(\)](#), and `offscreenTextureSampler`.

Referenced by [init\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.6 createPushConstantRange()

```
void PostStage::createPushConstantRange() [private]
```

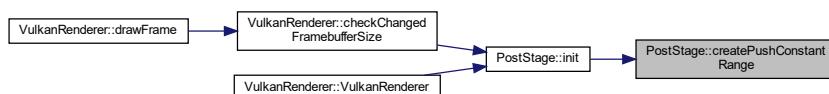
Definition at line 156 of file [PostStage.cpp](#).

```
00156     {
00157     push_constant_range.stageFlags =
00158         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
00159     push_constant_range.offset = 0;
00160     push_constant_range.size = sizeof(PushConstantPost);
00161 }
```

References [push\\_constant\\_range](#).

Referenced by [init\(\)](#).

Here is the caller graph for this function:



### 5.21.3.7 createRenderpass()

```
void PostStage::createRenderpass() [private]
```

Definition at line 163 of file [PostStage.cpp](#).

```
00163     {
00164     // Color attachment of render pass
00165     VkAttachmentDescription color_attachment{};
00166     const VKFormat& swap_chain_image_format =
00167         vulkanSwapChain->getSwapChainFormat();
00168     color_attachment.format =
00169         swap_chain_image_format; // format to use for attachment
00170     color_attachment.samples =
00171         VK_SAMPLE_COUNT_1_BIT; // number of samples to write for multisampling
00172     color_attachment.loadOp =
00173         VK_ATTACHMENT_LOAD_OP_CLEAR; // describes what to do with attachment
00174                                         // before rendering
00175     color_attachment.storeOp =
00176         VK_ATTACHMENT_STORE_OP_STORE; // describes what to do with attachment
00177                                         // after rendering
  
```

```

00178     color_attachment.stencilLoadOp =
00179         VK_ATTACHMENT_LOAD_OP_DONT_CARE; // describes what to do with stencil
00180                                     // before rendering
00181     color_attachment.stencilStoreOp =
00182         VK_ATTACHMENT_STORE_OP_DONT_CARE; // describes what to do with stencil
00183                                     // after rendering
00184
00185     // framebuffer data will be stored as an image, but images can be given
00186     // different layouts to give optimal use for certain operations
00187     color_attachment.initialLayout =
00188         VK_IMAGE_LAYOUT_UNDEFINED; // image data layout before render pass starts
00189     color_attachment.finalLayout =
00190         VK_IMAGE_LAYOUT_PRESENT_SRC_KHR; // image data layout after render pass
00191                                     // (to change to)
00192
00193     // depth attachment of render pass
00194     VkAttachmentDescription depth_attachment{};
00195     depth_attachment.format = choose_supported_format(
00196         device->getPhysicalDevice(),
00197         {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00198          VK_FORMAT_D24_UNORM_S8_UINT},
00199         VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00200     depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
00201     depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
00202     depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00203     depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
00204     depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00205     depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
00206     depth_attachment.finalLayout =
00207         VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00208
00209     // attachment reference uses an attachment index that refers to index in the
00210     // attachment list passed to renderPassCreateInfo
00211     VkAttachmentReference color_attachment_reference{};
00212     color_attachment_reference.attachment = 0;
00213     color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
00214
00215     // attachment reference
00216     VkAttachmentReference depth_attachment_reference{};
00217     depth_attachment_reference.attachment = 1;
00218     depth_attachment_reference.layout =
00219         VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00220
00221     // information about a particular subpass the render pass is using
00222     VkSubpassDescription subpass{};
00223     subpass.pipelineBindPoint =
00224         VK_PIPELINE_BIND_POINT_GRAPHICS; // pipeline type subpass is to be bound
00225                                     // to
00226     subpass.colorAttachmentCount = 1;
00227     subpass.pColorAttachments = &color_attachment_reference;
00228     subpass.pDepthStencilAttachment = &depth_attachment_reference;
00229
00230     // need to determine when layout transitions occur using subpass dependencies
00231     std::array<VkSubpassDependency, 1> subpass_dependencies;
00232
00233     // conversion from VK_IMAGE_LAYOUT_UNDEFINED to
00234     // VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
00235     subpass_dependencies[0].srcSubpass =
00236         VK_SUBPASS_EXTERNAL; // subpass index (VK_SUBPASS_EXTERNAL = Special
00237                                     // value meaning outside of renderpass)
00238     subpass_dependencies[0].srcStageMask =
00239         VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT; // pipeline stage
00240     subpass_dependencies[0].srcAccessMask =
00241         VK_ACCESS_MEMORY_READ_BIT; // stage access mask (memory access)
00242     subpass_dependencies[0].dstSubpass = 0;
00243     subpass_dependencies[0].dstStageMask =
00244         VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00245     subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
00246                                     VK_ACCESS_COLOR_ATTACHMENT_READ_BIT;
00247     subpass_dependencies[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
00248
00249     std::array<VkAttachmentDescription, 2> render_pass_attachments = {
00250         color_attachment, depth_attachment};
00251
00252     // create info for render pass
00253     VkRenderPassCreateInfo render_pass_create_info{};
00254     render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
00255     render_pass_create_info.attachmentCount =
00256         static_cast<uint32_t>(render_pass_attachments.size());
00257     render_pass_create_info.pAttachments = render_pass_attachments.data();
00258     render_pass_create_info.subpassCount = 1;
00259     render_pass_create_info.pSubpasses = &subpass;
00260     render_pass_create_info.dependencyCount =
00261         static_cast<uint32_t>(subpass_dependencies.size());
00262     render_pass_create_info.pDependencies = subpass_dependencies.data();
00263
00264     VkResult result =

```

```

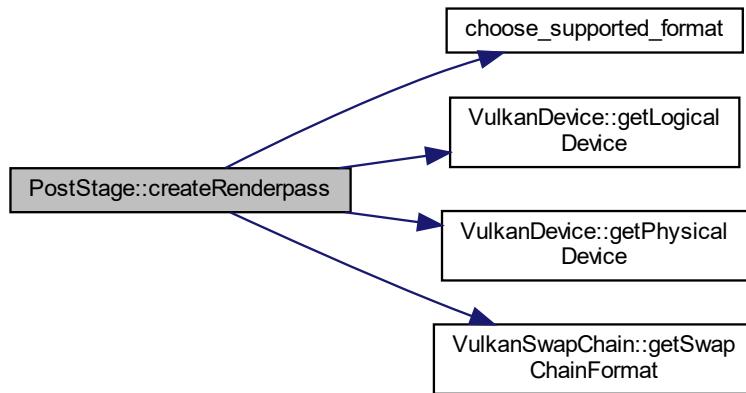
00265     vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
00266                           nullptr, &render_pass);
00267     ASSERT_VULKAN(result, "Failed to create render pass!")
00268 }

```

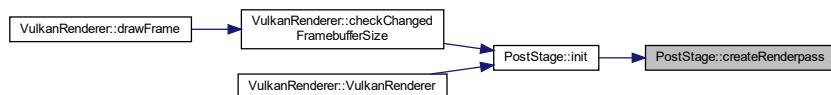
References [choose\\_supported\\_format\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanDevice::getPhysicalDevice\(\)](#), [VulkanSwapChain::getSwapChainFormat\(\)](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [init\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.8 getOffscreenSampler()

```
VkSampler & PostStage::getOffscreenSampler() [inline]
```

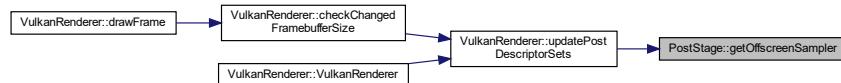
Definition at line 19 of file [PostStage.hpp](#).

```
00019 { return offscreenTextureSampler; };
```

References [offscreenTextureSampler](#).

Referenced by [VulkanRenderer::updatePostDescriptorSets\(\)](#).

Here is the caller graph for this function:



### 5.21.3.9 getRenderPass()

```
VkRenderPass & PostStage::getRenderPass ( ) [inline]
```

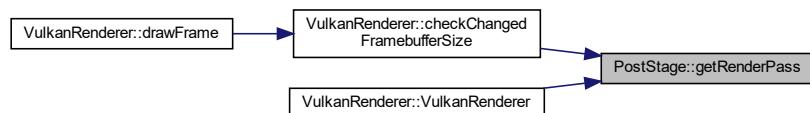
Definition at line 18 of file [PostStage.hpp](#).

```
00018 { return render_pass; };
```

References [render\\_pass](#).

Referenced by [VulkanRenderer::checkChangedFramebufferSize\(\)](#), and [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the caller graph for this function:



### 5.21.3.10 init()

```
void PostStage::init (
    VulkanDevice * device,
    VulkanSwapChain * vulkanSwapChain,
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts )
```

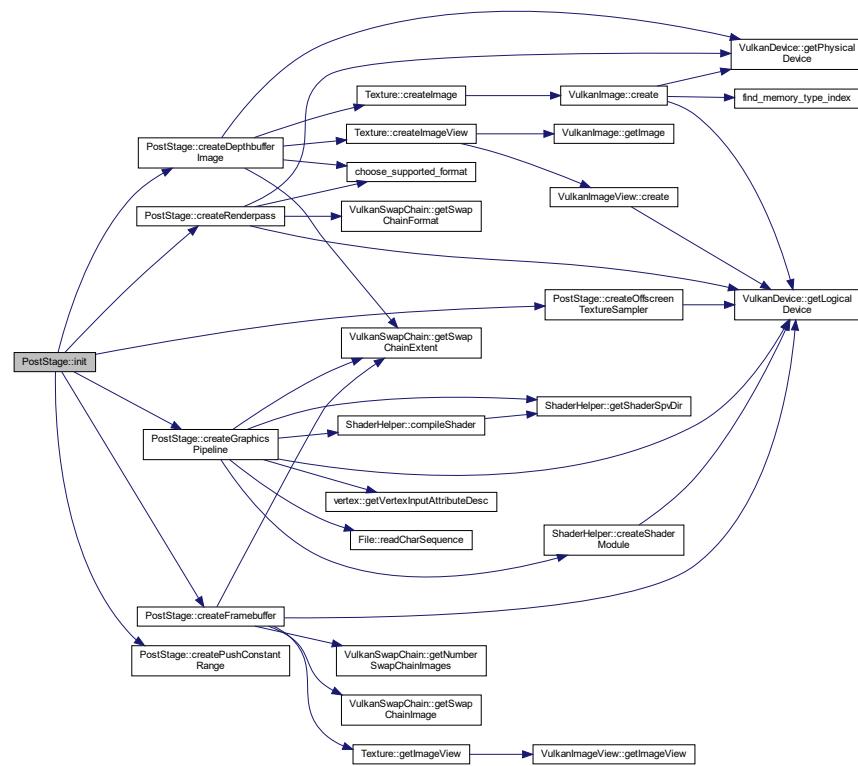
Definition at line 17 of file [PostStage.cpp](#).

```
00019 {
00020     this->device = device;
00021     this->vulkanSwapChain = vulkanSwapChain;
00022
00023     createOffscreenTextureSampler();
00024
00025     createPushConstantRange();
00026     createDepthbufferImage();
00027     createRenderpass();
00028     createGraphicsPipeline(descriptorSetLayouts);
00029     createFramebuffer();
00030 }
```

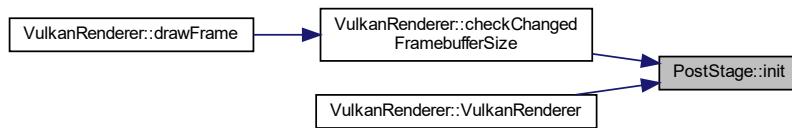
References `createDepthbufferImage()`, `createFramebuffer()`, `createGraphicsPipeline()`, `createOffscreenTextureSampler()`, `createPushConstantRange()`, `createRenderpass()`, `device`, and `vulkanSwapChain`.

Referenced by `VulkanRenderer::checkChangedFramebufferSize()`, and `VulkanRenderer::VulkanRenderer()`.

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.11 recordCommands()

```
void PostStage::recordCommands (
    VkCommandBuffer & commandBuffer,
```

```
    uint32_t image_index,
    const std::vector< VkDescriptorSet > & descriptorSets )
```

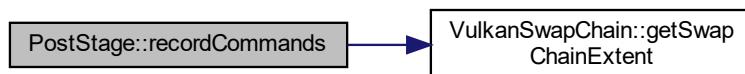
Definition at line 38 of file [PostStage.cpp](#).

```
00040
00041     // information about how to begin a render pass (only needed for graphical
00042     // applications)
00043     VkRenderPassBeginInfo render_pass_begin_info{};
00044     render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
00045     render_pass_begin_info.renderPass = render_pass; // render pass to begin
00046     render_pass_begin_info.renderArea.offset = {
00047         0, 0}; // start point of render pass in pixels
00048     const VKExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00049     render_pass_begin_info.renderArea.extent =
00050         swap_chain_extent; // size of region to run render pass on (starting at
00051                         // offset)
00052
00053     // make sure the order you put the values into the array matches with the
00054     // attachment order you have defined previous
00055     std::array<VkClearValue, 2> clear_values = {};
00056     clear_values[0].color = {0.2f, 0.65f, 0.4f, 1.0f};
00057     clear_values[1].depthStencil = {1.0f, 0};
00058
00059     render_pass_begin_info.pClearValues = clear_values.data();
00060     render_pass_begin_info.clearValueCount =
00061         static_cast<uint32_t>(clear_values.size());
00062
00063     // used framebuffer depends on the swap chain and therefore is changing for
00064     // each command buffer
00065     render_pass_begin_info.framebuffer = framebuffers[image_index];
00066
00067     // begin render pass
00068     vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
00069                           VK_SUBPASS_CONTENTS_INLINE);
00070     auto aspectRatio = static_cast<float>(swap_chain_extent.width) /
00071                         static_cast<float>(swap_chain_extent.height);
00072     PushConstantPost pc_post{};
00073     pc_post.aspect_ratio = aspectRatio;
00074     vkCmdPushConstants(commandBuffer, pipeline_layout,
00075                         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT,
00076                         0, sizeof(PushConstantPost), &pc_post);
00077     vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00078                         graphics_pipeline);
00079     vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00080                             pipeline_layout, 0,
00081                             static_cast<uint32_t>(descriptorSets.size()),
00082                             descriptorSets.data(), 0, nullptr);
00083     vkCmdDraw(commandBuffer, 3, 1, 0, 0);
00084
00085     // Rendering gui
00086     ImGui::Render();
00087     ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), commandBuffer);
00088
00089     // end render pass
00090     vkCmdEndRenderPass(commandBuffer);
00091 }
```

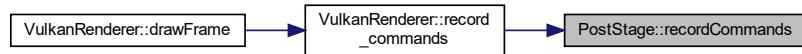
References [PushConstantPost::aspect\\_ratio](#), [framebuffers](#), [VulkanSwapChain::getSwapChainExtent\(\)](#), [graphics\\_pipeline](#), [pipeline\\_layout](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [VulkanRenderer::record\\_commands\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.21.3.12 shaderHotReload()

```
void PostStage::shaderHotReload (
    const std::vector< VkDescriptorSetLayout > & descriptor_set_layouts )
```

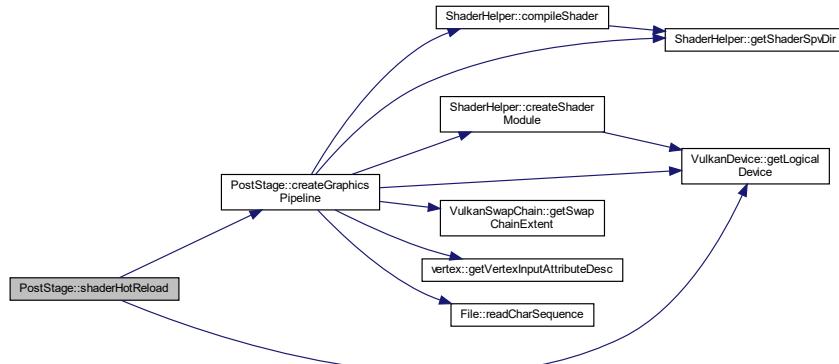
Definition at line 32 of file [PostStage.cpp](#).

```
00033
00034     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00035     createGraphicsPipeline(descriptor_set_layouts);
00036 }
```

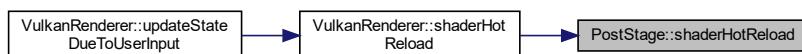
References [createGraphicsPipeline\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), and [graphics\\_pipeline](#).

Referenced by [VulkanRenderer::shaderHotReload\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.21.4 Field Documentation

### 5.21.4.1 depth\_format

```
VkFormat PostStage::depth_format {VK_FORMAT_UNDEFINED} [private]
```

Definition at line 33 of file [PostStage.hpp](#).

Referenced by [createDepthbufferImage\(\)](#).

### 5.21.4.2 depthBufferedImage

```
Texture PostStage::depthBufferedImage [private]
```

Definition at line 32 of file [PostStage.hpp](#).

Referenced by [cleanUp\(\)](#), [createDepthbufferImage\(\)](#), and [createFramebuffer\(\)](#).

### 5.21.4.3 device

```
VulkanDevice* PostStage::device {VK_NULL_HANDLE} [private]
```

Definition at line 28 of file [PostStage.hpp](#).

Referenced by [cleanUp\(\)](#), [createDepthbufferImage\(\)](#), [createFramebuffer\(\)](#), [createGraphicsPipeline\(\)](#), [createOffscreenTextureSampler\(\)](#), [createRenderpass\(\)](#), [init\(\)](#), and [shaderHotReload\(\)](#).

### 5.21.4.4 framebuffers

```
std::vector<VkFramebuffer> PostStage::framebuffers [private]
```

Definition at line 31 of file [PostStage.hpp](#).

Referenced by [cleanUp\(\)](#), [createFramebuffer\(\)](#), and [recordCommands\(\)](#).

#### 5.21.4.5 `graphics_pipeline`

```
VkPipeline PostStage::graphics_pipeline {VK_NULL_HANDLE} [private]
```

Definition at line 42 of file [PostStage.hpp](#).

Referenced by [cleanUp\(\)](#), [createGraphicsPipeline\(\)](#), [recordCommands\(\)](#), and [shaderHotReload\(\)](#).

#### 5.21.4.6 `offscreenTextureSampler`

```
VkSampler PostStage::offscreenTextureSampler [private]
```

Definition at line 36 of file [PostStage.hpp](#).

Referenced by [cleanUp\(\)](#), [createOffscreenTextureSampler\(\)](#), and [getOffscreenSampler\(\)](#).

#### 5.21.4.7 `pipeline_layout`

```
VkPipelineLayout PostStage::pipeline_layout {VK_NULL_HANDLE} [private]
```

Definition at line 43 of file [PostStage.hpp](#).

Referenced by [cleanUp\(\)](#), [createGraphicsPipeline\(\)](#), and [recordCommands\(\)](#).

#### 5.21.4.8 `push_constant_range`

```
VkPushConstantRange PostStage::push_constant_range [private]
```

##### **Initial value:**

```
{VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0,  
 0}
```

Definition at line 39 of file [PostStage.hpp](#).

Referenced by [createGraphicsPipeline\(\)](#), and [createPushConstantRange\(\)](#).

#### 5.21.4.9 `render_pass`

```
VkRenderPass PostStage::render_pass {VK_NULL_HANDLE} [private]
```

Definition at line 41 of file [PostStage.hpp](#).

Referenced by [cleanUp\(\)](#), [createFramebuffer\(\)](#), [createGraphicsPipeline\(\)](#), [createRenderpass\(\)](#), [getRenderPass\(\)](#), and [recordCommands\(\)](#).

### 5.21.4.10 vulkanSwapChain

```
VulkanSwapChain* PostStage::vulkanSwapChain {VK_NULL_HANDLE} [private]
```

Definition at line 29 of file [PostStage.hpp](#).

Referenced by [createDepthbufferImage\(\)](#), [createFramebuffer\(\)](#), [createGraphicsPipeline\(\)](#), [createRenderpass\(\)](#), [init\(\)](#), and [recordCommands\(\)](#).

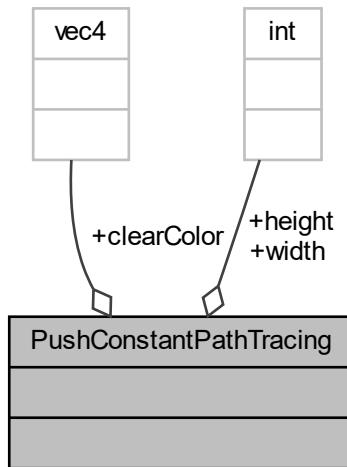
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[PostStage.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/[PostStage.cpp](#)

## 5.22 PushConstantPathTracing Struct Reference

```
#include <PushConstantPathTracing.hpp>
```

Collaboration diagram for PushConstantPathTracing:



### Data Fields

- `vec4 clearColor`
- `uint width`
- `uint height`

### 5.22.1 Detailed Description

Definition at line 17 of file [PushConstantPathTracing.hpp](#).

## 5.22.2 Field Documentation

### 5.22.2.1 clearColor

```
vec4 PushConstantPathTracing::clearColor
```

Definition at line 18 of file [PushConstantPathTracing.hpp](#).

Referenced by [PathTracing::recordCommands\(\)](#).

### 5.22.2.2 height

```
uint PushConstantPathTracing::height
```

Definition at line 20 of file [PushConstantPathTracing.hpp](#).

Referenced by [PathTracing::recordCommands\(\)](#).

### 5.22.2.3 width

```
uint PushConstantPathTracing::width
```

Definition at line 19 of file [PushConstantPathTracing.hpp](#).

Referenced by [PathTracing::recordCommands\(\)](#).

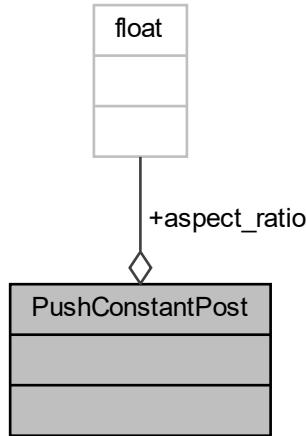
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/[PushConstantPathTracing.hpp](#)

## 5.23 PushConstantPost Struct Reference

```
#include <PushConstantPost.hpp>
```

Collaboration diagram for PushConstantPost:



### Data Fields

- float `aspect_ratio`

#### 5.23.1 Detailed Description

Definition at line 17 of file [PushConstantPost.hpp](#).

#### 5.23.2 Field Documentation

##### 5.23.2.1 `aspect_ratio`

```
float PushConstantPost::aspect_ratio
```

Definition at line 18 of file [PushConstantPost.hpp](#).

Referenced by [PostStage::recordCommands\(\)](#).

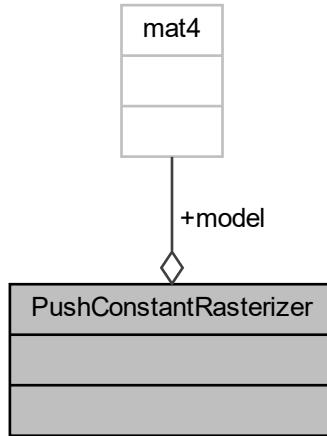
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/[PushConstantPost.hpp](#)

## 5.24 PushConstantRasterizer Struct Reference

```
#include <PushConstantRasterizer.hpp>
```

Collaboration diagram for PushConstantRasterizer:



### Data Fields

- [mat4 model](#)

#### 5.24.1 Detailed Description

Definition at line 18 of file [PushConstantRasterizer.hpp](#).

#### 5.24.2 Field Documentation

##### 5.24.2.1 model

```
mat4 PushConstantRasterizer::model
```

Definition at line 19 of file [PushConstantRasterizer.hpp](#).

Referenced by [Rasterizer::recordCommands\(\)](#).

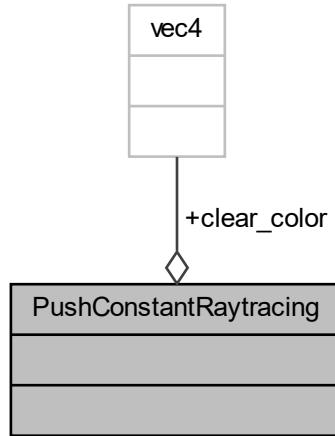
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/[PushConstantRasterizer.hpp](#)

## 5.25 PushConstantRaytracing Struct Reference

```
#include <PushConstantRayTracing.hpp>
```

Collaboration diagram for PushConstantRaytracing:



### Data Fields

- `vec4 clear_color`

#### 5.25.1 Detailed Description

Definition at line 17 of file [PushConstantRayTracing.hpp](#).

#### 5.25.2 Field Documentation

##### 5.25.2.1 clear\_color

```
vec4 PushConstantRaytracing::clear_color
```

Definition at line 18 of file [PushConstantRayTracing.hpp](#).

Referenced by [Raytracing::recordCommands\(\)](#).

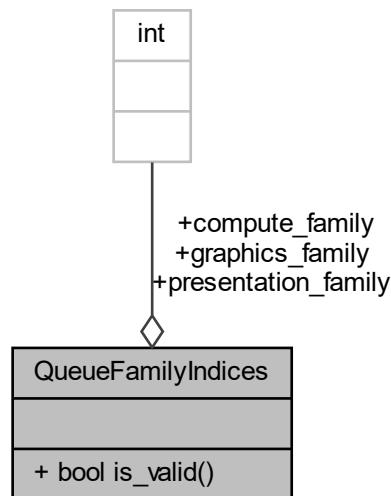
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/[PushConstantRayTracing.hpp](#)

## 5.26 QueueFamilyIndices Struct Reference

```
#include <QueueFamilyIndices.hpp>
```

Collaboration diagram for QueueFamilyIndices:



### Public Member Functions

- `bool is_valid ()`

### Data Fields

- `int graphics_family = -1`
- `int presentation_family = -1`
- `int compute_family = -1`

#### 5.26.1 Detailed Description

Definition at line 3 of file [QueueFamilyIndices.hpp](#).

#### 5.26.2 Member Function Documentation

### 5.26.2.1 is\_valid()

```
bool QueueFamilyIndices::is_valid () [inline]
```

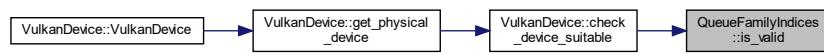
Definition at line 9 of file [QueueFamilyIndices.hpp](#).

```
00009     {  
00010         return graphics_family >= 0 && presentation_family >= 0 &&  
00011             compute_family >= 0;  
00012     }
```

References [compute\\_family](#), [graphics\\_family](#), and [presentation\\_family](#).

Referenced by [VulkanDevice::check\\_device\\_suitable\(\)](#).

Here is the caller graph for this function:



## 5.26.3 Field Documentation

### 5.26.3.1 compute\_family

```
int QueueFamilyIndices::compute_family = -1
```

Definition at line 6 of file [QueueFamilyIndices.hpp](#).

Referenced by [VulkanRenderer::create\\_command\\_pool\(\)](#), [VulkanDevice::create\\_logical\\_device\(\)](#), [is\\_valid\(\)](#), and [PathTracing::recordCommands\(\)](#).

### 5.26.3.2 graphics\_family

```
int QueueFamilyIndices::graphics_family = -1
```

Definition at line 4 of file [QueueFamilyIndices.hpp](#).

Referenced by [VulkanRenderer::create\\_command\\_pool\(\)](#), [GUI::create\\_gui\\_context\(\)](#), [VulkanDevice::create\\_logical\\_device\(\)](#), [VulkanSwapChain::initVulkanContext\(\)](#), [is\\_valid\(\)](#), and [PathTracing::recordCommands\(\)](#).

### 5.26.3.3 presentation\_family

```
int QueueFamilyIndices::presentation_family = -1;
```

Definition at line 5 of file [QueueFamilyIndices.hpp](#).

Referenced by `VulkanDevice::create_logical_device()`, `VulkanSwapChain::initVulkanContext()`, and `is_valid()`.

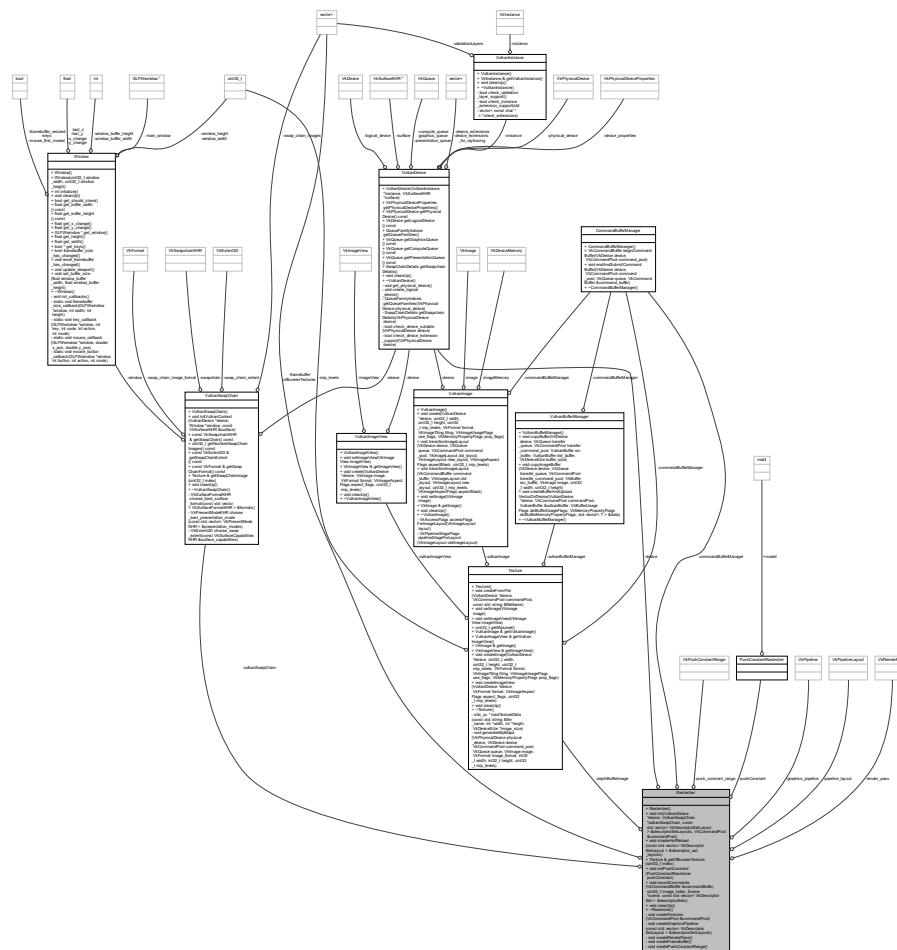
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_16e3q/Desktop/GraphicsEngineVulkan/include/renderer/QueueFamilyIndices.hpp

## 5.27 Rasterizer Class Reference

```
#include <Rasterizer.hpp>
```

## Collaboration diagram for Rasterizer:



## Public Member Functions

- `Rasterizer ()`
- `void init (VulkanDevice *device, VulkanSwapChain *vulkanSwapChain, const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts, VkCommandPool &commandPool)`
- `void shaderHotReload (const std::vector< VkDescriptorSetLayout > &descriptor_set_layouts)`
- `Texture & getOffscreenTexture (uint32_t index)`
- `void setPushConstant (PushConstantRasterizer pushConstant)`
- `void recordCommands (VkCommandBuffer &commandBuffer, uint32_t image_index, Scene *scene, const std::vector< VkDescriptorSet > &descriptorSets)`
- `void cleanUp ()`
- `~Rasterizer ()`

## Private Member Functions

- `void createTextures (VkCommandPool &commandPool)`
- `void createGraphicsPipeline (const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts)`
- `void createRenderPass ()`
- `void createFramebuffer ()`
- `void createPushConstantRange ()`

## Private Attributes

- `VulkanDevice * device {VK_NULL_HANDLE}`
- `VulkanSwapChain * vulkanSwapChain {VK_NULL_HANDLE}`
- `CommandBufferManager commandBufferManager`
- `std::vector< VkFramebuffer > framebuffer`
- `std::vector< Texture > offscreenTextures`
- `Texture depthBufferImage`
- `VkPushConstantRange push_constant_range`
- `PushConstantRasterizer pushConstant {glm::mat4(1.f)}`
- `VkPipeline graphics_pipeline {VK_NULL_HANDLE}`
- `VkPipelineLayout pipeline_layout {VK_NULL_HANDLE}`
- `VkRenderPass render_pass {VK_NULL_HANDLE}`

### 5.27.1 Detailed Description

Definition at line 10 of file [Rasterizer.hpp](#).

### 5.27.2 Constructor & Destructor Documentation

#### 5.27.2.1 Rasterizer()

```
Rasterizer::Rasterizer ( )
```

Definition at line 14 of file [Rasterizer.cpp](#).  
00014 { }

### 5.27.2.2 ~Rasterizer()

```
Rasterizer::~Rasterizer ( )
```

Definition at line 132 of file [Rasterizer.cpp](#).

```
00132 { }
```

## 5.27.3 Member Function Documentation

### 5.27.3.1 cleanUp()

```
void Rasterizer::cleanUp ( )
```

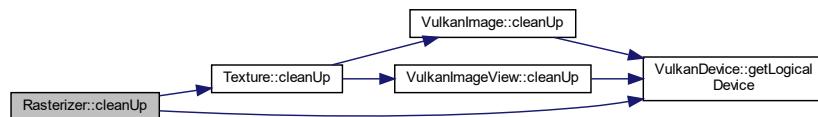
Definition at line 116 of file [Rasterizer.cpp](#).

```
00116 {
00117     for (auto framebuffer : framebuffer) {
00118         vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
00119     }
00120     for (Texture texture : offscreenTextures) {
00121         texture.cleanUp();
00122     }
00123     depthBufferImage.cleanUp();
00124
00125     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00126     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00127     vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
00128
00129 }
00130 }
```

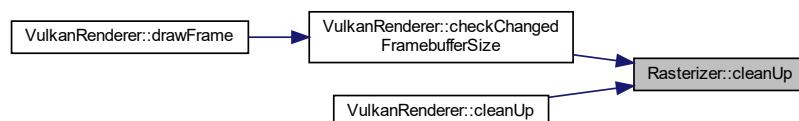
References [Texture::cleanUp\(\)](#), [depthBufferImage](#), [device](#), [framebuffer](#), [VulkanDevice::getLogicalDevice\(\)](#), [graphics\\_pipeline](#), [offscreenTextures](#), [pipeline\\_layout](#), and [render\\_pass](#).

Referenced by [VulkanRenderer::checkChangedFramebufferSize\(\)](#), and [VulkanRenderer::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.27.3.2 createFramebuffer()

```
void Rasterizer::createFramebuffer () [private]
```

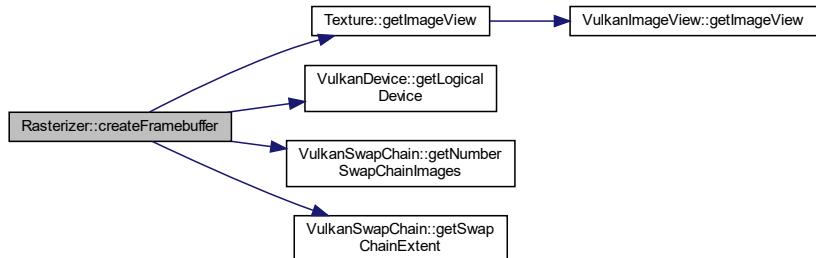
Definition at line 243 of file [Rasterizer.cpp](#).

```
00243     {
00244         framebuffer.resize(vulkanSwapChain->getNumberSwapChainImages());
00245
00246         for (size_t i = 0; i < framebuffer.size(); i++) {
00247             std::array<VkImageView, 2> attachments = {
00248                 offscreenTextures[i].getImageView(), depthBufferImage.getImageView()};
00249
00250             VkFramebufferCreateInfo frame_buffer_create_info{};
00251             frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
00252             frame_buffer_create_info.renderPass = render_pass;
00253             frame_buffer_create_info.attachmentCount =
00254                 static_cast<uint32_t>(attachments.size());
00255             frame_buffer_create_info.pAttachments = attachments.data();
00256             const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00257             frame_buffer_create_info.width = swap_chain_extent.width;
00258             frame_buffer_create_info.height = swap_chain_extent.height;
00259             frame_buffer_create_info.layers = 1;
00260
00261             VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
00262                                                 &frame_buffer_create_info, nullptr,
00263                                                 &framebuffer[i]);
00264             ASSERT_VULKAN(result, "Failed to create framebuffer!");
00265         }
00266     }
```

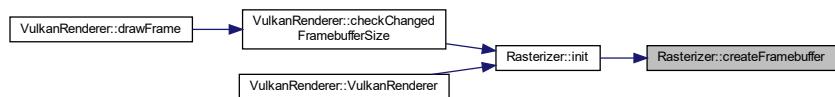
References [depthBufferImage](#), [device](#), [framebuffer](#), [Texture::getImageView\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanSwapChain::getNumberSwapChainImages\(\)](#), [VulkanSwapChain::getSwapChainExtent\(\)](#), [offscreenTextures](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [init\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.27.3.3 createGraphicsPipeline()

```

void Rasterizer::createGraphicsPipeline (
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts ) [private]

Definition at line 343 of file Rasterizer.cpp.
00344     std::stringstream rasterizer_shader_dir;
00345     std::filesystem::path cwd = std::filesystem::current_path();
00346     rasterizer_shader_dir << cwd.string();
00347     rasterizer_shader_dir << RELATIVE_RESOURCE_PATH;
00348     rasterizer_shader_dir << "Shaders/rasterizer/";
00349
00350
00351     ShaderHelper shaderHelper;
00352     shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.vert");
00353     shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.frag");
00354
00355     File vertexFile(
00356         shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.vert"));
00357     File fragmentFile(
00358         shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.frag"));
00359     std::vector<char> vertex_shader_code = vertexFile.readCharSequence();
00360     std::vector<char> fragment_shader_code = fragmentFile.readCharSequence();
00361
00362     // build shader modules to link to graphics pipeline
00363     VkShaderModule vertex_shader_module =
00364         shaderHelper.createShaderModule(device, vertex_shader_code);
00365     VkShaderModule fragment_shader_module =
00366         shaderHelper.createShaderModule(device, fragment_shader_code);
00367
00368     // shader stage creation information
00369     // vertex stage creation information
00370     VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
00371     vertex_shader_create_info.sType =
00372         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00373     vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
00374     vertex_shader_create_info.module = vertex_shader_module;
00375     vertex_shader_create_info.pName = "main";
00376
00377     // fragment stage creation information
00378     VkPipelineShaderStageCreateInfo fragment_shader_create_info{};
00379     fragment_shader_create_info.sType =
00380         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00381     fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
00382     fragment_shader_create_info.module = fragment_shader_module;
00383     fragment_shader_create_info.pName = "main";
00384
00385     std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
00386         vertex_shader_create_info, fragment_shader_create_info};
00387
00388     // how the data for a single vertex (including info such as position, color,
00389     // texture coords, normals, etc) is as a whole
00390     VkVertexInputBindingDescription binding_description{};
00391     binding_description.binding = 0;
00392     binding_description.stride = sizeof(Vertex);
00393     binding_description.inputRate =
00394         VK_VERTEX_INPUT_RATE_VERTEX; // how to move between data after each
00395         // vertex.
00396
00397     // how the data for an attribute is defined within a vertex
00398     std::array<VkVertexInputAttributeDescription, 4> attribute_descriptions =
00399         vertex::getVertexInputAttributeDesc();
00400
00401     // CREATE PIPELINE
00402     // 1.) Vertex input
00403     VkPipelineVertexInputStateCreateInfo vertex_input_create_info{};
00404     vertex_input_create_info.sType =
00405         VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
00406     vertex_input_create_info.vertexBindingDescriptionCount = 1;
00407     vertex_input_create_info.pVertexBindingDescriptions = &binding_description;
00408     vertex_input_create_info.vertexAttributeDescriptionCount =
00409         static_cast<uint32_t>(attribute_descriptions.size());
00410     vertex_input_create_info.pVertexAttributeDescriptions =
00411         attribute_descriptions.data();
00412
00413     // input assembly
00414     VkPipelineInputAssemblyStateCreateInfo input_assembly{};
00415     input_assembly.sType =
00416         VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
00417     input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
00418     input_assembly.primitiveRestartEnable = VK_FALSE;
00419
00420     // viewport & scissor
00421     // create a viewport info struct

```

```

00422     VkViewport viewport{};
00423     viewport.x = 0.0f;
00424     viewport.y = 0.0f;
00425     const VKExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00426     viewport.width = (float)swap_chain_extent.width;
00427     viewport.height = (float)swap_chain_extent.height;
00428     viewport.minDepth = 0.0f;
00429     viewport.maxDepth = 1.0f;
00430
00431     // create a scissor info struct
00432     VkRect2D scissor{};
00433     scissor.offset = {0, 0};
00434     scissor.extent = swap_chain_extent;
00435
00436     VkPipelineViewportStateCreateInfo viewport_state_create_info{};
00437     viewport_state_create_info.sType =
00438         VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
00439     viewport_state_create_info.viewportCount = 1;
00440     viewport_state_create_info.pViewports = &viewport;
00441     viewport_state_create_info.scissorCount = 1;
00442     viewport_state_create_info.pScissors = &scissor;
00443
00444     // RASTERIZER
00445     VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
00446     rasterizer_create_info.sType =
00447         VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
00448     rasterizer_create_info.depthClampEnable = VK_FALSE;
00449     rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
00450     rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
00451     rasterizer_create_info.lineWidth = 1.0f;
00452     rasterizer_create_info.cullMode = VK_CULL_MODE_BACK_BIT;    //
00453     // winding to determine which side is front; y-coordinate is inverted in
00454     // comparison to OpenGL
00455     rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
00456     rasterizer_create_info.depthBiasClamp = VK_FALSE;
00457
00458     // -- MULTISAMPLING --
00459     VkPipelineMultisampleStateCreateInfo multisample_create_info{};
00460     multisample_create_info.sType =
00461         VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
00462     multisample_create_info.sampleShadingEnable = VK_FALSE;
00463     multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
00464
00465     // -- BLENDING --
00466     // blend attachment state
00467     VkPipelineColorBlendAttachmentState color_state{};
00468     color_state.colorWriteMask =
00469         VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
00470         VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
00471
00472     color_state.blendEnable = VK_TRUE;
00473     // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
00474     // (dstColorBlendFactor * old_color)
00475     color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
00476     color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
00477     color_state.colorBlendOp = VK_BLEND_OP_ADD;
00478     color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
00479     color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
00480     color_state.alphaBlendOp = VK_BLEND_OP_ADD;
00481
00482     VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
00483     color_blending_create_info.sType =
00484         VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
00485     color_blending_create_info.logicOpEnable = VK_FALSE;
00486     color_blending_create_info.attachmentCount = 1;
00487     color_blending_create_info.pAttachments = &color_state;
00488
00489     // -- PIPELINE LAYOUT --
00490     VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00491     pipeline_layout_create_info.sType =
00492         VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00493     pipeline_layout_create_info.setLayoutCount =
00494         static_cast<uint32_t>(descriptorSetLayouts.size());
00495     pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00496     pipeline_layout_create_info.pushConstantRangeCount = 1;
00497     pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;
00498
00499     // create pipeline layout
00500     VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00501                                             &pipeline_layout_create_info,
00502                                             nullptr, &pipeline_layout);
00503     ASSERT_VULKAN(result, "Failed to create pipeline layout!")
00504
00505     // -- DEPTH_STENCIL TESTING --
00506     VkPipelineDepthStencilStateCreateInfo depth_stencil_create_info{};
00507     depth_stencil_create_info.sType =
00508         VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;

```

```

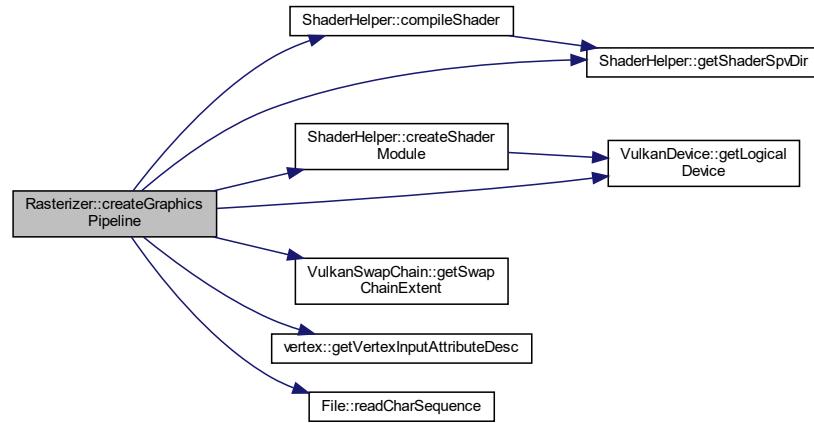
00509 depth_stencil_create_info.depthTestEnable = VK_TRUE;
00510 depth_stencil_create_info.depthWriteEnable = VK_TRUE;
00511 depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS;
00512 depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
00513 depth_stencil_create_info.stencilTestEnable = VK_FALSE;
00514
00515 // -- GRAPHICS PIPELINE CREATION --
00516 VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
00517 graphics_pipeline_create_info.sType =
00518     VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
00519 graphics_pipeline_create_info.stageCount =
00520     static_cast<uint32_t>(shader_stages.size());
00521 graphics_pipeline_create_info.pStages = shader_stages.data();
00522 graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
00523 graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
00524 graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
00525 graphics_pipeline_create_info.pDynamicState = nullptr;
00526 graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
00527 graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
00528 graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
00529 graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
00530 graphics_pipeline_create_info.layout = pipeline_layout;
00531 graphics_pipeline_create_info.renderPass = render_pass;
00532 graphics_pipeline_create_info.subpass = 0;
00533
00534 // pipeline derivatives : can create multiple pipelines that derive from one
00535 // another for optimization
00536 graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
00537 graphics_pipeline_create_info.basePipelineIndex = -1;
00538
00539 // create graphics pipeline
00540 result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
00541                                     1, &graphics_pipeline_create_info, nullptr,
00542                                     &graphics_pipeline);
00543 ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")
00544
00545 // Destroy shader modules, no longer needed after pipeline created
00546 vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
00547                         nullptr);
00548 vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,
00549                         nullptr);
00550 }

```

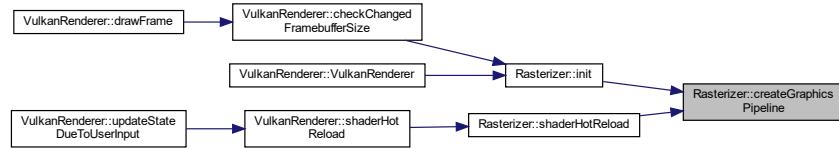
References [ShaderHelper::compileShader\(\)](#), [ShaderHelper::createShaderModule\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [ShaderHelper::getShaderSpvDir\(\)](#), [VulkanSwapChain::getSwapChainExtent\(\)](#), [vertex::getVertexInputAttributeDesc\(\)](#), [graphics\\_pipeline](#), [pipeline\\_layout](#), [push\\_constant\\_range](#), [File::readCharSequence\(\)](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [init\(\)](#), and [shaderHotReload\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.27.3.4 createPushConstantRange()

```
void Rasterizer::createPushConstantRange() [private]
```

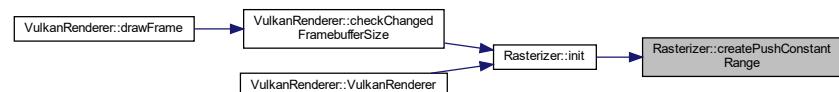
Definition at line 268 of file [Rasterizer.cpp](#).

```
00268     {
00269     // define push constant values (no 'create' needed)
00270     push_constant_range.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
00271     push_constant_range.offset = 0;
00272     push_constant_range.size = sizeof(PushConstantRasterizer);
00273 }
```

References [push\\_constant\\_range](#).

Referenced by [init\(\)](#).

Here is the caller graph for this function:



#### 5.27.3.5 createRenderPass()

```
void Rasterizer::createRenderPass() [private]
```

Definition at line 134 of file [Rasterizer.cpp](#).

```
00134     {
00135     // Color attachment of render pass
00136     VkAttachmentDescription color_attachment{};
00137     const VkFormat& swap_chain_image_format =
00138         vulkanSwapChain->getSwapChainFormat();
00139     color_attachment.format =
00140         swap_chain_image_format; // format to use for attachment
00141     color_attachment.samples =
00142         VK_SAMPLE_COUNT_1_BIT; // number of samples to write for multisampling
00143     color_attachment.loadOp =
00144         VK_ATTACHMENT_LOAD_OP_CLEAR; // describes what to do with attachment
00145                                         // before rendering
00146     color_attachment.storeOp =
```

```

00147     VK_ATTACHMENT_STORE_OP_STORE; // describes what to do with attachment
00148                                         // after rendering
00149     color_attachment.stencilLoadOp =
00150         VK_ATTACHMENT_LOAD_OP_DONT_CARE; // describes what to do with stencil
00151                                         // before rendering
00152     color_attachment.stencilStoreOp =
00153         VK_ATTACHMENT_STORE_OP_DONT_CARE; // describes what to do with stencil
00154                                         // after rendering
00155
00156     // framebuffer data will be stored as an image, but images can be given
00157     // different layouts to give optimal use for certain operations
00158     color_attachment.initialLayout =
00159         VK_IMAGE_LAYOUT_GENERAL; // image data layout before render pass starts
00160     color_attachment.finalLayout =
00161         VK_IMAGE_LAYOUT_GENERAL; // image data layout after render pass (to
00162                                         // change to)
00163
00164     // depth attachment of render pass
00165     VkAttachmentDescription depth_attachment{};
00166     depth_attachment.format = choose_supported_format(
00167         device->getPhysicalDevice(),
00168         {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00169          VK_FORMAT_D24_UNORM_S8_UINT},
00170         VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00171
00172     depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
00173     depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
00174     depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00175     depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
00176     depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00177     depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
00178     depth_attachment.finalLayout =
00179         VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00180
00181     // attachment reference uses an attachment index that refers to index in the
00182     // attachment list passed to renderPassCreateInfo
00183     VkAttachmentReference color_attachment_reference{};
00184     color_attachment_reference.attachment = 0;
00185     color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
00186
00187     // attachment reference
00188     VkAttachmentReference depth_attachment_reference{};
00189     depth_attachment_reference.attachment = 1;
00190     depth_attachment_reference.layout =
00191         VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00192
00193     // information about a particular subpass the render pass is using
00194     VkSubpassDescription subpass{};
00195     subpass.pipelineBindPoint =
00196         VK_PIPELINE_BIND_POINT_GRAPHICS; // pipeline type subpass is to be bound
00197                                         // to
00198     subpass.colorAttachmentCount = 1;
00199     subpass.pColorAttachments = &color_attachment_reference;
00200     subpass.pDepthStencilAttachment = &depth_attachment_reference;
00201
00202     // need to determine when layout transitions occur using subpass dependencies
00203     std::array<VkSubpassDependency, 1> subpass_dependencies;
00204
00205     // conversion from VK_IMAGE_LAYOUT_UNDEFINED to
00206     // VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
00207     subpass_dependencies[0].srcSubpass =
00208         VK_SUBPASS_EXTERNAL; // subpass index (VK_SUBPASS_EXTERNAL = Special
00209                                         // value meaning outside of renderpass)
00210     subpass_dependencies[0].srcStageMask =
00211         VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT; // pipeline stage
00212     subpass_dependencies[0].srcAccessMask =
00213         0; // stage access mask (memory access)
00214
00215     // but must happen before ...
00216     subpass_dependencies[0].dstSubpass = 0;
00217     subpass_dependencies[0].dstStageMask =
00218         VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00219     subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
00220     subpass_dependencies[0].dependencyFlags = 0; // VK_DEPENDENCY_BY_REGION_BIT;
00221
00222     std::array<VkAttachmentDescription, 2> render_pass_attachments = {
00223         color_attachment, depth_attachment};
00224
00225     // create info for render pass
00226     VkRenderPassCreateInfo render_pass_create_info{};
00227     render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
00228     render_pass_create_info.attachmentCount =
00229         static_cast<uint32_t>(render_pass_attachments.size());
00230     render_pass_create_info.pAttachments = render_pass_attachments.data();
00231     render_pass_create_info.subpassCount = 1;
00232     render_pass_create_info.pSubpasses = &subpass;
00233     render_pass_create_info.dependencyCount =

```

```

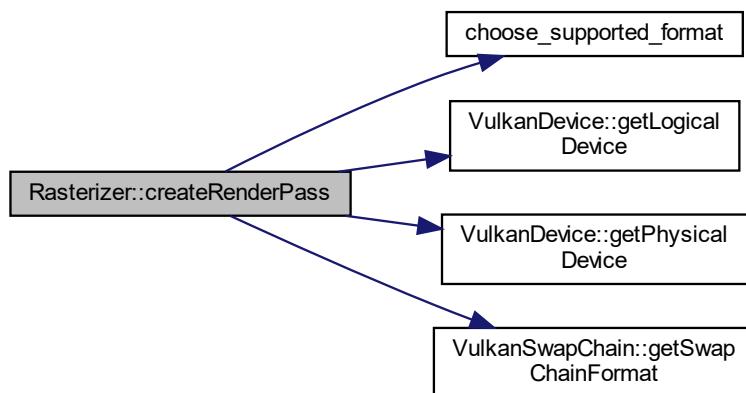
00234     static_cast<uint32_t>(subpass_dependencies.size());
00235     render_pass_create_info.pDependencies = subpass_dependencies.data();
00236
00237     VkResult result =
00238         vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
00239                             nullptr, &render_pass);
00240     ASSERT_VULKAN(result, "Failed to create render pass!")
00241 }

```

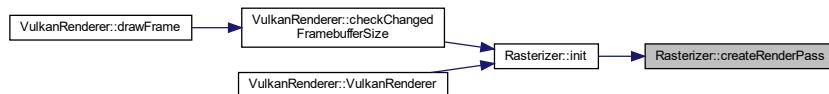
References [choose\\_supported\\_format\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanDevice::getPhysicalDevice\(\)](#), [VulkanSwapChain::getSwapChainFormat\(\)](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [init\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.27.3.6 createTextures()

```

void Rasterizer::createTextures (
    VkCommandPool & commandPool ) [private]

```

Definition at line 275 of file [Rasterizer.cpp](#).

```

00275     {
00276         offscreenTextures.resize(vulkanSwapChain->getNumberSwapChainImages());
00277
00278         VkCommandBuffer cmdBuffer = commandBufferManager.beginCommandBuffer(
00279             device->getLogicalDevice(), commandPool);

```

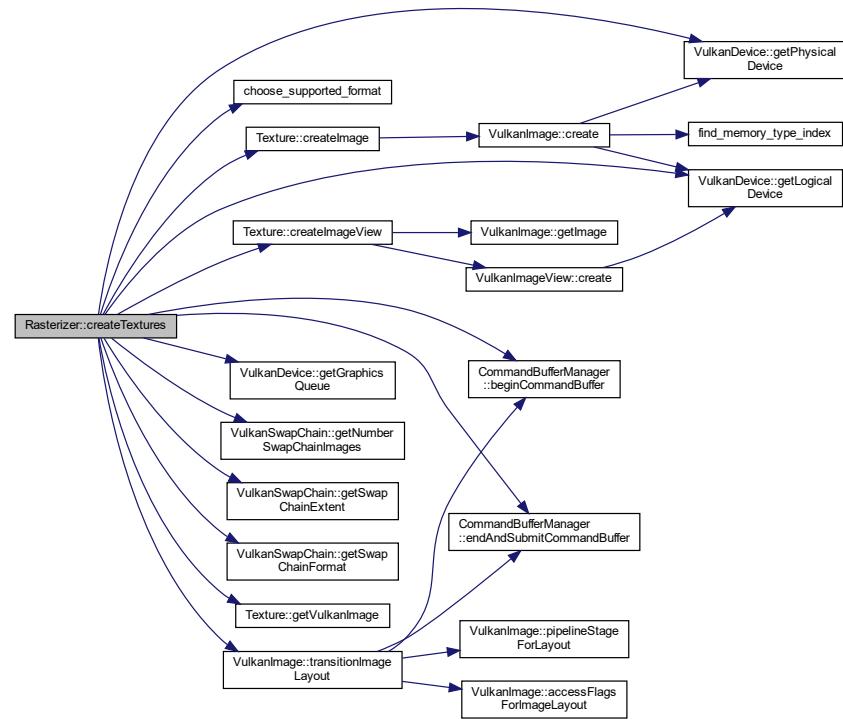
```

00280
00281     for (uint32_t index = 0;
00282         index < static_cast<uint32_t>(vulkanSwapChain->getNumberSwapChainImages());
00283         index++) {
00284         Texture texture{};
00285         const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00286         const VkFormat& swap_chain_image_format =
00287             vulkanSwapChain->getSwapChainFormat();
00288
00289         texture.createImage(
00290             device, swap_chain_extent.width, swap_chain_extent.height, 1,
00291             swap_chain_image_format, VK_IMAGE_TILING_OPTIMAL,
00292             VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT |
00293                 VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT,
00294                 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00295
00296         texture.createImageView(device, swap_chain_image_format,
00297             VK_IMAGE_ASPECT_COLOR_BIT, 1);
00298
00299         // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
00300         VulkanImage& image = texture.getVulkanImage();
00301         image.transitionImageLayout(cmdBuffer, VK_IMAGE_LAYOUT_UNDEFINED,
00302             VK_IMAGE_LAYOUT_GENERAL, 1,
00303             VK_IMAGE_ASPECT_COLOR_BIT);
00304
00305         offscreenTextures[index] = texture;
00306     }
00307
00308     VkFormat depth_format = choose_supported_format(
00309         device->getPhysicalDevice(),
00310         {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00311             VK_FORMAT_D24_UNORM_S8_UINT},
00312         VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00313
00314
00315     // create depth buffer image
00316     // MIP LEVELS: for depth texture we only want 1 level :)
00317     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00318     depthBufferImage.createImage(device, swap_chain_extent.width,
00319         swap_chain_extent.height, 1, depth_format,
00320         VK_IMAGE_TILING_OPTIMAL,
00321         VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
00322         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00323
00324
00325     // depth buffer image view
00326     // MIP LEVELS: for depth texture we only want 1 level :)
00327     depthBufferImage.createImageView(
00328         device, depth_format,
00329         VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);
00330
00331     // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
00332     VulkanImage& vulkanImage = depthBufferImage.getVulkanImage();
00333     vulkanImage.transitionImageLayout(
00334         device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
00335         VK_IMAGE_LAYOUT_UNDEFINED,
00336         VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
00337         VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);
00338
00339     commandBufferManager.endAndSubmitCommandBuffer(
00340         device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
00341         cmdBuffer);
00342
00343 }
```

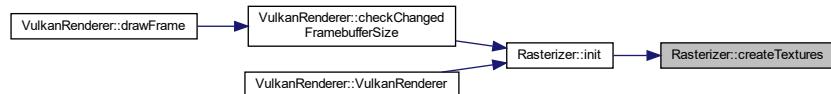
References `CommandBufferManager::beginCommandBuffer()`, `choose_supported_format()`, `commandBufferManager`, `Texture::createImage()`, `Texture::createImageView()`, `depthBufferImage`, `device`, `CommandBufferManager::endAndSubmitCommandBuffer()`, `VulkanDevice::getGraphicsQueue()`, `VulkanDevice::getLogicalDevice()`, `VulkanSwapChain::getNumberSwapChainImages()`, `VulkanDevice::getPhysicalDevice()`, `VulkanSwapChain::getSwapChainExtent()`, `VulkanSwapChain::getSwapChainFormat()`, `Texture::getVulkanImage()`, `offscreenTextures`, `VulkanImage::transitionImageLayout()`, and `vulkanSwapChain`.

Referenced by `init()`.

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.27.3.7 `getOffscreenTexture()`

```
Texture & Rasterizer::getOffscreenTexture (
    uint32_t index )
```

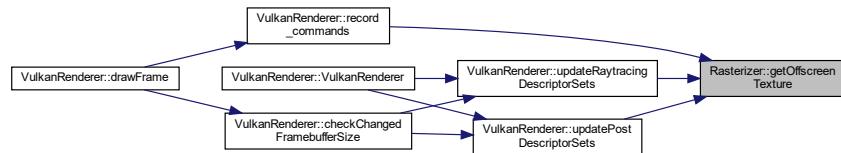
Definition at line 36 of file `Rasterizer.cpp`.

```
00036
00037     return offscreenTextures[index];
00038 }
```

References [offscreenTextures](#).

Referenced by `VulkanRenderer::record_commands()`, `VulkanRenderer::updatePostDescriptorSets()`, and `VulkanRenderer::updateRaytracingDescriptorSets()`.

Here is the caller graph for this function:



### 5.27.3.8 init()

```

void Rasterizer::init (
    VulkanDevice * device,
    VulkanSwapChain * vulkanSwapChain,
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts,
    VkCommandPool & commandPool )
  
```

Definition at line 16 of file `Rasterizer.cpp`.

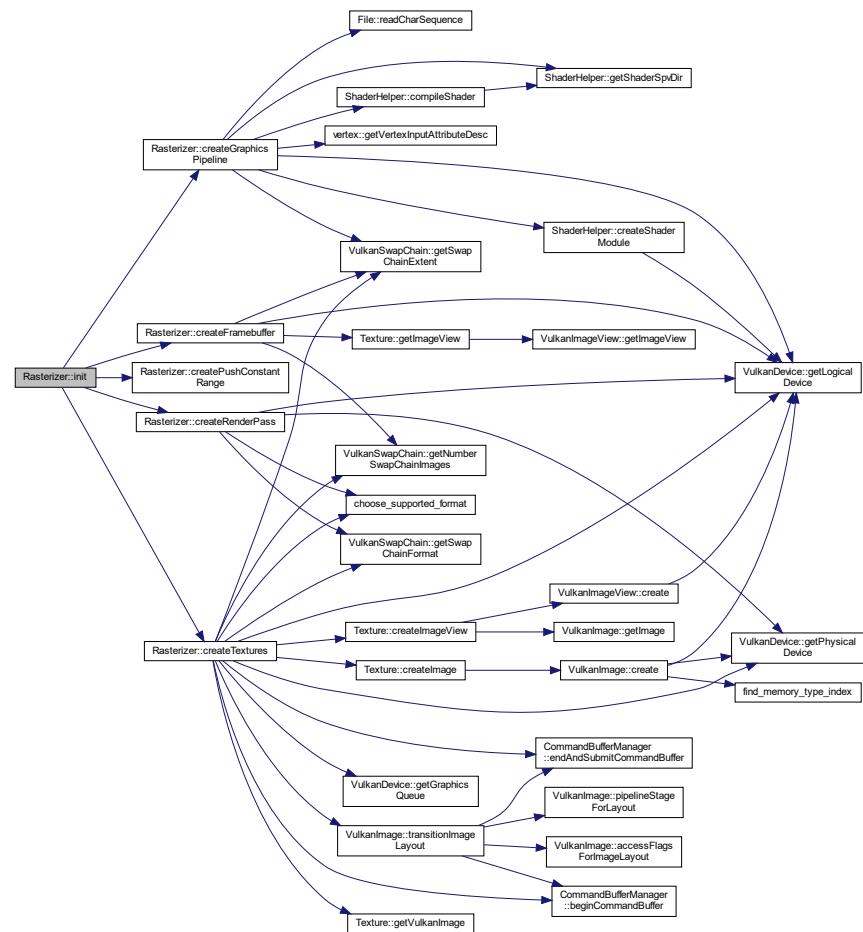
```

00019     this->device = device;
00020     this->vulkanSwapChain = vulkanSwapChain;
00022
00023     createTextures(commandPool);
00024     createRenderPass();
00025     createPushConstantRange();
00026     createGraphicsPipeline(descriptorSetLayouts);
00027     createFramebuffer();
00028 }
  
```

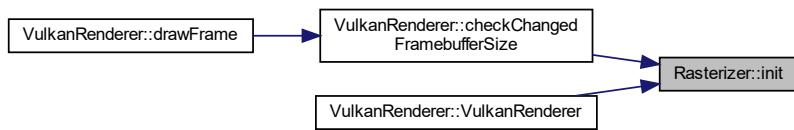
References `createFramebuffer()`, `createGraphicsPipeline()`, `createPushConstantRange()`, `createRenderPass()`, `createTextures()`, `device`, and `vulkanSwapChain`.

Referenced by `VulkanRenderer::checkChangedFramebufferSize()`, and `VulkanRenderer::VulkanRenderer()`.

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.27.3.9 recordCommands()

```
void Rasterizer::recordCommands (
    VkCommandBuffer & commandBuffer,
    uint32_t imageIndex,
```

```
Scene * scene,
const std::vector< VkDescriptorSet > & descriptorSets )
```

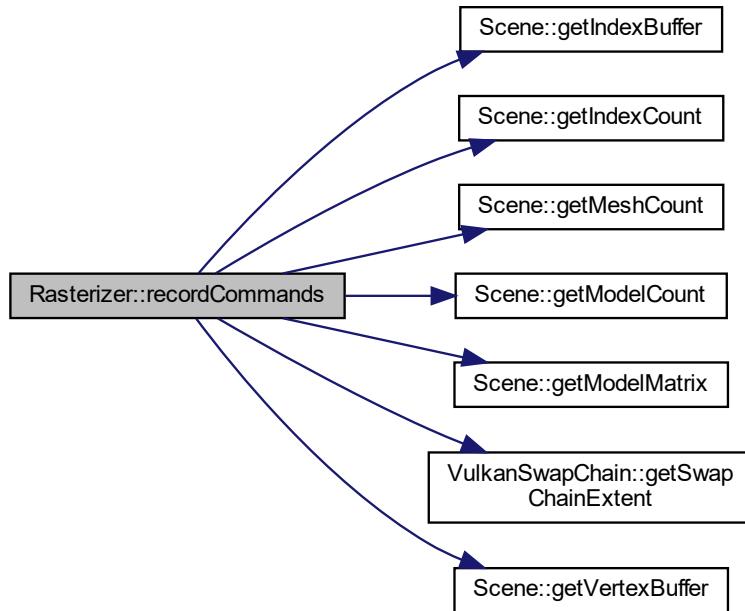
Definition at line 44 of file [Rasterizer.cpp](#).

```
00046
00047     // information about how to begin a render pass (only needed for graphical
00048     // applications)
00049     VkRenderPassBeginInfo render_pass_begin_info{};
00050     render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
00051     render_pass_begin_info.renderPass = render_pass;
00052     render_pass_begin_info.renderArea.offset = {0, 0};
00053     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00054     render_pass_begin_info.renderArea.extent = swap_chain_extent;
00055
00056     // make sure the order you put the values into the array matches with the
00057     // attachment order you have defined previous
00058     std::array<VkClearValue, 2> clear_values = {};
00059     clear_values[0].color = {0.2f, 0.65f, 0.4f, 1.0f};
00060     clear_values[1].depthStencil = {1.0f, 0};
00061
00062     render_pass_begin_info.pClearValues = clear_values.data();
00063     render_pass_begin_info.clearValueCount =
00064         static_cast<uint32_t>(clear_values.size());
00065     render_pass_begin_info.framebuffer = framebuffer[image_index];
00066
00067     // begin render pass
00068     vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
00069                           VK_SUBPASS_CONTENTS_INLINE);
00070
00071     // bind pipeline to be used in render pass
00072     vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00073                        graphics_pipeline);
00074
00075     for (uint32_t m = 0; m < static_cast<uint32_t>(scene->getModelCount()); m++) {
00076         // for GCC doesn't allow references on rvalues go like that ...
00077         pushConstant.model = scene->getModelMatrix(0);
00078         // just "Push" constants to given shader stage directly (no buffer)
00079         vkCmdPushConstants(
00080             commandBuffer, pipeline_layout,
00081             VK_SHADER_STAGE_VERTEX_BIT,           // stage to push constants to
00082             0,                                // offset to push constants to update
00083             sizeof(PushConstantRasterizer),      // size of data being pushed
00084             &pushConstant); // using model of current mesh (can be array)
00085
00086         for (unsigned int k = 0; k < scene->getMeshCount(m); k++) {
00087             // list of vertex buffers we want to draw
00088             VkBuffer vertex_buffers[] = {
00089                 scene->getVertexBuffer(m, k)}; // buffers to bind
00090             VkDeviceSize offsets[] = {0};
00091             vkCmdBindVertexBuffers(
00092                 commandBuffer, 0, 1, vertex_buffers,
00093                 offsets); // command to bind vertex buffer before drawing with them
00094
00095             // bind mesh index buffer with 0 offset and using the uint32 type
00096             vkCmdBindIndexBuffer(commandBuffer, scene->getIndexBuffer(m, k), 0,
00097                                 VK_INDEX_TYPE_UINT32);
00098
00099             // bind descriptor sets
00100             vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00101                                     pipeline_layout, 0,
00102                                     static_cast<uint32_t>(descriptorSets.size()),
00103                                     descriptorSets.data(), 0, nullptr);
00104
00105             // execute pipeline
00106             vkCmdDrawIndexed(commandBuffer,
00107                             static_cast<uint32_t>(scene->getIndexCount(m, k)), 1, 0,
00108                             0, 0);
00109         }
00110     }
00111
00112     // end render pass
00113     vkCmdEndRenderPass(commandBuffer);
00114 }
```

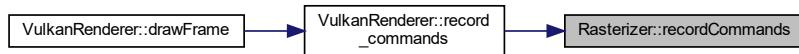
References [framebuffer](#), [Scene::getIndexBuffer\(\)](#), [Scene::getIndexCount\(\)](#), [Scene::getMeshCount\(\)](#), [Scene::getModelCount\(\)](#), [Scene::getModelMatrix\(\)](#), [VulkanSwapChain::getSwapChainExtent\(\)](#), [Scene::getVertexBuffer\(\)](#), [graphics\\_pipeline](#), [PushConstantRasterizer::model](#), [pipeline\\_layout](#), [pushConstant](#), [render\\_pass](#), and [vulkanSwapChain](#).

Referenced by [VulkanRenderer::record\\_commands\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.27.3.10 `setPushConstant()`

```
void Rasterizer::setPushConstant (
    PushConstantRasterizer pushConstant )
```

Definition at line 40 of file `Rasterizer.cpp`.

```
00040
00041     this->pushConstant = pushConstant;
00042 }
```

References [pushConstant](#).

### 5.27.3.11 shaderHotReload()

```
void Rasterizer::shaderHotReload (
    const std::vector< VkDescriptorSetLayout > & descriptor_set_layouts )
```

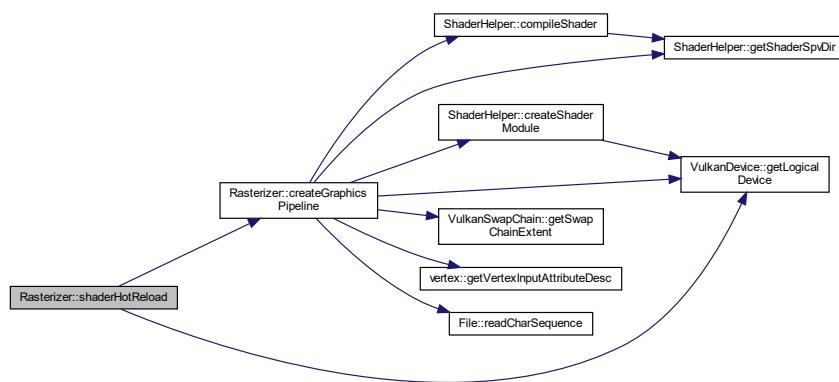
Definition at line 30 of file [Rasterizer.cpp](#).

```
00031     {
00032     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00033     createGraphicsPipeline(descriptor_set_layouts);
00034 }
```

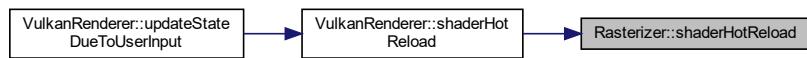
References [createGraphicsPipeline\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), and [graphics\\_pipeline](#).

Referenced by [VulkanRenderer::shaderHotReload\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.27.4 Field Documentation

### 5.27.4.1 commandBufferManager

```
CommandBufferManager Rasterizer::commandBufferManager [private]
```

Definition at line 37 of file [Rasterizer.hpp](#).

Referenced by [createTextures\(\)](#).

#### 5.27.4.2 depthBufferedImage

```
Texture Rasterizer::depthBufferedImage [private]
```

Definition at line 41 of file [Rasterizer.hpp](#).

Referenced by [cleanUp\(\)](#), [createFramebuffer\(\)](#), and [createTextures\(\)](#).

#### 5.27.4.3 device

```
VulkanDevice* Rasterizer::device {VK_NULL_HANDLE} [private]
```

Definition at line 34 of file [Rasterizer.hpp](#).

Referenced by [cleanUp\(\)](#), [createFramebuffer\(\)](#), [createGraphicsPipeline\(\)](#), [createRenderPass\(\)](#), [createTextures\(\)](#), [init\(\)](#), and [shaderHotReload\(\)](#).

#### 5.27.4.4 framebuffer

```
std::vector<VkFramebuffer> Rasterizer::framebuffer [private]
```

Definition at line 39 of file [Rasterizer.hpp](#).

Referenced by [cleanUp\(\)](#), [createFramebuffer\(\)](#), and [recordCommands\(\)](#).

#### 5.27.4.5 graphics\_pipeline

```
VkPipeline Rasterizer::graphics_pipeline {VK_NULL_HANDLE} [private]
```

Definition at line 47 of file [Rasterizer.hpp](#).

Referenced by [cleanUp\(\)](#), [createGraphicsPipeline\(\)](#), [recordCommands\(\)](#), and [shaderHotReload\(\)](#).

#### 5.27.4.6 offscreenTextures

```
std::vector<Texture> Rasterizer::offscreenTextures [private]
```

Definition at line 40 of file [Rasterizer.hpp](#).

Referenced by [cleanUp\(\)](#), [createFramebuffer\(\)](#), [createTextures\(\)](#), and [getOffscreenTexture\(\)](#).

#### 5.27.4.7 pipeline\_layout

```
VkPipelineLayout Rasterizer::pipeline_layout {VK_NULL_HANDLE} [private]
```

Definition at line 48 of file [Rasterizer.hpp](#).

Referenced by [cleanUp\(\)](#), [createGraphicsPipeline\(\)](#), and [recordCommands\(\)](#).

#### 5.27.4.8 push\_constant\_range

```
VkPushConstantRange Rasterizer::push_constant_range [private]
```

##### Initial value:

```
{VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0,
          0}
```

Definition at line 43 of file [Rasterizer.hpp](#).

Referenced by [createGraphicsPipeline\(\)](#), and [createPushConstantRange\(\)](#).

#### 5.27.4.9 pushConstant

```
PushConstantRasterizer Rasterizer::pushConstant {glm::mat4(1.f)} [private]
```

Definition at line 45 of file [Rasterizer.hpp](#).

Referenced by [recordCommands\(\)](#), and [setPushConstant\(\)](#).

#### 5.27.4.10 render\_pass

```
VkRenderPass Rasterizer::render_pass {VK_NULL_HANDLE} [private]
```

Definition at line 49 of file [Rasterizer.hpp](#).

Referenced by [cleanUp\(\)](#), [createFramebuffer\(\)](#), [createGraphicsPipeline\(\)](#), [createRenderPass\(\)](#), and [recordCommands\(\)](#).

#### 5.27.4.11 vulkanSwapChain

```
VulkanSwapChain* Rasterizer::vulkanSwapChain {VK_NULL_HANDLE} [private]
```

Definition at line 35 of file [Rasterizer.hpp](#).

Referenced by [createFramebuffer\(\)](#), [createGraphicsPipeline\(\)](#), [createRenderPass\(\)](#), [createTextures\(\)](#), [init\(\)](#), and [recordCommands\(\)](#).

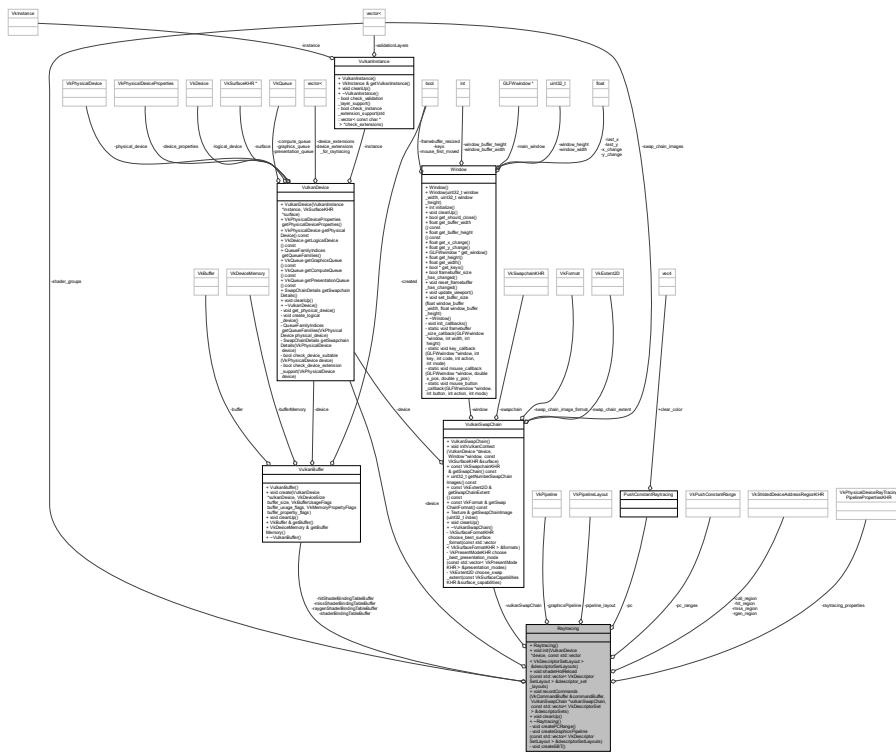
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[Rasterizer.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/[Rasterizer.cpp](#)

## 5.28 Raytracing Class Reference

```
#include <Raytracing.hpp>
```

## Collaboration diagram for Raytracing:



## Public Member Functions

- `Raytracing ()`
  - `void init (VulkanDevice *device, const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts)`
  - `void shaderHotReload (const std::vector< VkDescriptorSetLayout > &descriptor_set_layouts)`
  - `void recordCommands (VkCommandBuffer &commandBuffer, VulkanSwapChain *vulkanSwapChain, const std::vector< VkDescriptorSet > &descriptorSets)`
  - `void cleanUp ()`
  - `~Raytracing ()`

## Private Member Functions

- void `createPCRange ()`
  - void `createGraphicsPipeline (const std::vector< VkDescriptorSetLayout > &descriptorSetLayouts)`
  - void `createSBT ()`

## Private Attributes

- `VulkanDevice * device {VK_NULL_HANDLE}`
- `VulkanSwapChain * vulkanSwapChain {VK_NULL_HANDLE}`
- `VkPipeline graphicsPipeline {VK_NULL_HANDLE}`
- `VkPipelineLayout pipeline_layout {VK_NULL_HANDLE}`
- `PushConstantRaytracing pc {glm::vec4(0.f)}`
- `VkPushConstantRange pc_ranges {VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0, 0}`
- `std::vector< VkRayTracingShaderGroupCreateInfoKHR > shader_groups`
- `VulkanBuffer shaderBindingTableBuffer`
- `VulkanBuffer raygenShaderBindingTableBuffer`
- `VulkanBuffer missShaderBindingTableBuffer`
- `VulkanBuffer hitShaderBindingTableBuffer`
- `VkStridedDeviceAddressRegionKHR rgen_region {}`
- `VkStridedDeviceAddressRegionKHR miss_region {}`
- `VkStridedDeviceAddressRegionKHR hit_region {}`
- `VkStridedDeviceAddressRegionKHR call_region {}`
- `VkPhysicalDeviceRayTracingPipelinePropertiesKHR raytracing_properties {}`

### 5.28.1 Detailed Description

Definition at line 9 of file [Raytracing.hpp](#).

### 5.28.2 Constructor & Destructor Documentation

#### 5.28.2.1 Raytracing()

`Raytracing::Raytracing ( )`

Definition at line 12 of file [Raytracing.cpp](#).  
00012 {}

#### 5.28.2.2 ~Raytracing()

`Raytracing::~Raytracing ( )`

Definition at line 99 of file [Raytracing.cpp](#).  
00099 {}

### 5.28.3 Member Function Documentation

### 5.28.3.1 cleanUp()

```
void Raytracing::cleanUp ( )
```

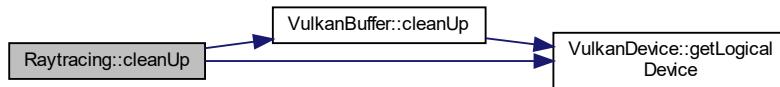
Definition at line 89 of file [Raytracing.cpp](#).

```
00089     {
00090     shaderBindingTableBuffer.cleanUp();
00091     raygenShaderBindingTableBuffer.cleanUp();
00092     missShaderBindingTableBuffer.cleanUp();
00093     hitShaderBindingTableBuffer.cleanUp();
00094
00095     vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
00096     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00097 }
```

References [VulkanBuffer::cleanUp\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [graphicsPipeline](#), [hitShaderBindingTableBuffer](#), [missShaderBindingTableBuffer](#), [pipeline\\_layout](#), [raygenShaderBindingTableBuffer](#), and [shaderBindingTableBuffer](#).

Referenced by [VulkanRenderer::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.28.3.2 createGraphicsPipeline()

```
void Raytracing::createGraphicsPipeline (
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts ) [private]
```

Definition at line 110 of file [Raytracing.cpp](#).

```
00111     {
00112     PFN_vkCreateRayTracingPipelinesKHR pvkCreateRayTracingPipelinesKHR =
00113         (PFN_vkCreateRayTracingPipelinesKHR)vkGetDeviceProcAddr(
00114             device->getLogicalDevice(), "vkCreateRayTracingPipelinesKHR");
00115
00116     std::stringstream raytracing_shader_dir;
00117     std::filesystem::path cwd = std::filesystem::current_path();
00118     raytracing_shader_dir << cwd.string();
```

```

00119 raytracing_shader_dir < RELATIVE_RESOURCE_PATH;
00120 raytracing_shader_dir < "Shaders/raytracing/";
00121
00122 std::string raygen_shader = "raytrace.rgen";
00123 std::string chit_shader = "raytrace.rchit";
00124 std::string miss_shader = "raytrace.rmiss";
00125 std::string shadow_shader = "shadow.rmiss";
00126
00127 ShaderHelper shaderHelper;
00128 shaderHelper.compileShader(raytracing_shader_dir.str(), raygen_shader);
00129 shaderHelper.compileShader(raytracing_shader_dir.str(), chit_shader);
00130 shaderHelper.compileShader(raytracing_shader_dir.str(), miss_shader);
00131 shaderHelper.compileShader(raytracing_shader_dir.str(), shadow_shader);
00132
00133 File raygenFile(
00134     shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), raygen_shader));
00135 File raychitFile(
00136     shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), chit_shader));
00137 File raymissFile(
00138     shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), miss_shader));
00139 File shadowFile(
00140     shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), shadow_shader));
00141
00142 std::vector<char> raygen_shader_code = raygenFile.readCharSequence();
00143 std::vector<char> raychit_shader_code = raychitFile.readCharSequence();
00144 std::vector<char> raymiss_shader_code = raymissFile.readCharSequence();
00145 std::vector<char> shadow_shader_code = shadowFile.readCharSequence();
00146
00147 // build shader modules to link to graphics pipeline
00148 VkShaderModule raygen_shader_module =
00149     shaderHelper.createShaderModule(device, raygen_shader_code);
00150 VkShaderModule raychit_shader_module =
00151     shaderHelper.createShaderModule(device, raychit_shader_code);
00152 VkShaderModule raymiss_shader_module =
00153     shaderHelper.createShaderModule(device, raymiss_shader_code);
00154 VkShaderModule shadow_shader_module =
00155     shaderHelper.createShaderModule(device, shadow_shader_code);
00156
00157 // create all shader stage infos for creating a group
00158 VkPipelineShaderStageCreateInfo rgen_shader_stage_info{};
00159 rgen_shader_stage_info.sType =
00160     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00161 rgen_shader_stage_info.stage = VK_SHADER_STAGE_RAYGEN_BIT_KHR;
00162 rgen_shader_stage_info.module = raygen_shader_module;
00163 rgen_shader_stage_info.pName = "main";
00164
00165 VkPipelineShaderStageCreateInfo rmiss_shader_stage_info{};
00166 rmiss_shader_stage_info.sType =
00167     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00168 rmiss_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
00169 rmiss_shader_stage_info.module = raymiss_shader_module;
00170 rmiss_shader_stage_info.pName = "main";
00171
00172 VkPipelineShaderStageCreateInfo shadow_shader_stage_info{};
00173 shadow_shader_stage_info.sType =
00174     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00175 shadow_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
00176 shadow_shader_stage_info.module = shadow_shader_module;
00177 shadow_shader_stage_info.pName = "main";
00178
00179 VkPipelineShaderStageCreateInfo rchit_shader_stage_info{};
00180 rchit_shader_stage_info.sType =
00181     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00182 rchit_shader_stage_info.stage = VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR;
00183 rchit_shader_stage_info.module = raychit_shader_module;
00184 rchit_shader_stage_info.pName = "main";
00185
00186 // we have all shader stages together
00187 std::array<VkPipelineShaderStageCreateInfo, 4> shader_stages = {
00188     rgen_shader_stage_info, rmiss_shader_stage_info, shadow_shader_stage_info,
00189     rchit_shader_stage_info};
00190
00191 enum StageIndices { eRaygen, eMiss, eMiss2, eClosestHit, eShaderGroupCount };
00192
00193 shader_groups.reserve(4);
00194 VkRayTracingShaderGroupCreateInfoKHR shader_group_create_infos[4];
00195
00196 shader_group_create_infos[0].sType =
00197     VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00198 shader_group_create_infos[0].pNext = nullptr;
00199 shader_group_create_infos[0].type =
00200     VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00201 shader_group_create_infos[0].generalShader = eRaygen;
00202 shader_group_create_infos[0].closestHitShader = VK_SHADER_UNUSED_KHR;
00203 shader_group_create_infos[0].anyHitShader = VK_SHADER_UNUSED_KHR;
00204 shader_group_create_infos[0].intersectionShader = VK_SHADER_UNUSED_KHR;
00205 shader_group_create_infos[0].pShaderGroupCaptureReplayHandle = nullptr;

```

```

00206
00207     shader_groups.push_back(shader_group_create_infos[0]);
00208
00209     shader_group_create_infos[1].sType =
00210         VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00211     shader_group_create_infos[1].pNext = nullptr;
00212     shader_group_create_infos[1].type =
00213         VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00214     shader_group_create_infos[1].generalShader = eMiss;
00215     shader_group_create_infos[1].closestHitShader = VK_SHADER_UNUSED_KHR;
00216     shader_group_create_infos[1].anyHitShader = VK_SHADER_UNUSED_KHR;
00217     shader_group_create_infos[1].intersectionShader = VK_SHADER_UNUSED_KHR;
00218     shader_group_create_infos[1].pShaderGroupCaptureReplayHandle = nullptr;
00219
00220     shader_groups.push_back(shader_group_create_infos[1]);
00221
00222     shader_group_create_infos[2].sType =
00223         VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00224     shader_group_create_infos[2].pNext = nullptr;
00225     shader_group_create_infos[2].type =
00226         VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00227     shader_group_create_infos[2].generalShader = eMiss2;
00228     shader_group_create_infos[2].closestHitShader = VK_SHADER_UNUSED_KHR;
00229     shader_group_create_infos[2].anyHitShader = VK_SHADER_UNUSED_KHR;
00230     shader_group_create_infos[2].intersectionShader = VK_SHADER_UNUSED_KHR;
00231     shader_group_create_infos[2].pShaderGroupCaptureReplayHandle = nullptr;
00232
00233     shader_groups.push_back(shader_group_create_infos[2]);
00234
00235     shader_group_create_infos[3].sType =
00236         VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00237     shader_group_create_infos[3].pNext = nullptr;
00238     shader_group_create_infos[3].type =
00239         VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR;
00240     shader_group_create_infos[3].generalShader = VK_SHADER_UNUSED_KHR;
00241     shader_group_create_infos[3].closestHitShader = eClosestHit;
00242     shader_group_create_infos[3].anyHitShader = VK_SHADER_UNUSED_KHR;
00243     shader_group_create_infos[3].intersectionShader = VK_SHADER_UNUSED_KHR;
00244     shader_group_create_infos[3].pShaderGroupCaptureReplayHandle = nullptr;
00245
00246     shader_groups.push_back(shader_group_create_infos[3]);
00247
00248     VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00249     pipeline_layout_create_info.sType =
00250         VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00251     pipeline_layout_create_info.setLayoutCount =
00252         static_cast<uint32_t>(descriptorSetLayouts.size());
00253     pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00254     pipeline_layout_create_info.pushConstantRangeCount = 1;
00255     pipeline_layout_create_info.pPushConstantRanges = &pc_ranges;
00256
00257     VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00258                                             &pipeline_layout_create_info,
00259                                             nullptr, &pipeline_layout);
00260     ASSERT_VULKAN(result, "Failed to create raytracing pipeline layout!")
00261
00262     VkPipelineLibraryCreateInfoKHR pipeline_library_create_info{};
00263     pipeline_library_create_info.sType =
00264         VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR;
00265     pipeline_library_create_info.pNext = nullptr;
00266     pipeline_library_create_info.libraryCount = 0;
00267     pipeline_library_create_info.pLibraries = nullptr;
00268
00269     VkRayTracingPipelineCreateInfoKHR raytracing_pipeline_create_info{};
00270     raytracing_pipeline_create_info.sType =
00271         VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR;
00272     raytracing_pipeline_create_info.pNext = nullptr;
00273     raytracing_pipeline_create_info.flags = 0;
00274     raytracing_pipeline_create_info.stageCount =
00275         static_cast<uint32_t>(shader_stages.size());
00276     raytracing_pipeline_create_info.pStages = shader_stages.data();
00277     raytracing_pipeline_create_info.groupCount =
00278         static_cast<uint32_t>(shader_groups.size());
00279     raytracing_pipeline_create_info.pGroups = shader_groups.data();
00280     /*raytracing_pipeline_create_info.pLibraryInfo =
00281         &pipeline_library_create_info;
00282         raytracing_pipeline_create_info.pLibraryInterface = NULL; */
00283     // TODO: HARDCODED FOR NOW;
00284     raytracing_pipeline_create_info.maxPipelineRayRecursionDepth = 2;
00285     raytracing_pipeline_create_info.layout = pipeline_layout;
00286
00287     result = pvkCreateRayTracingPipelinesKHR(
00288         device->getLogicalDevice(), VK_NULL_HANDLE, VK_NULL_HANDLE, 1,
00289         &raytracing_pipeline_create_info, nullptr, &graphicsPipeline);
00290
00291     ASSERT_VULKAN(result, "Failed to create raytracing pipeline!")
00292

```

```

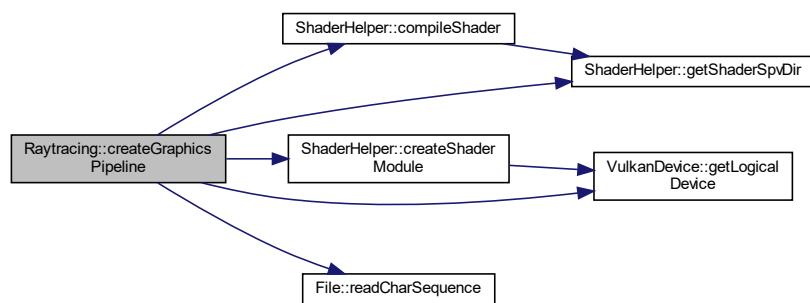
00293     vkDestroyShaderModule(device->getLogicalDevice(), raygen_shader_module,
00294                             nullptr);
00295     vkDestroyShaderModule(device->getLogicalDevice(), raymiss_shader_module,
00296                             nullptr);
00297     vkDestroyShaderModule(device->getLogicalDevice(), raychit_shader_module,
00298                             nullptr);
00299     vkDestroyShaderModule(device->getLogicalDevice(), shadow_shader_module,
00300                             nullptr);
00301 }

```

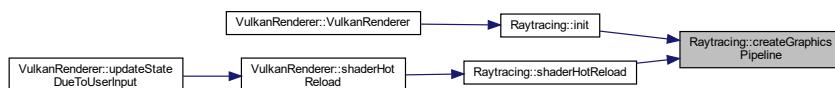
References [ShaderHelper::compileShader\(\)](#), [ShaderHelper::createShaderModule\(\)](#), [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [ShaderHelper::getShaderSpvDir\(\)](#), [graphicsPipeline](#), [pc\\_ranges](#), [pipeline\\_layout](#), [File::readCharSequence\(\)](#), and [shader\\_groups](#).

Referenced by [init\(\)](#), and [shaderHotReload\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.28.3.3 createPCRange()

```
void Raytracing::createPCRange( ) [private]
```

Definition at line 101 of file [Raytracing.cpp](#).

```

00101 {
00102     // define push constant values (no 'create' needed)
00103     pc_ranges.stageFlags = VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00104                             VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00105                             VK_SHADER_STAGE_MISS_BIT_KHR;
00106     pc_ranges.offset = 0;
00107     pc_ranges.size = sizeof(PushConstantRaytracing); // size of data being passed
00108 }

```

References [pc\\_ranges](#).

Referenced by [init\(\)](#).

Here is the caller graph for this function:



#### 5.28.3.4 createSBT()

```
void Raytracing::createSBT ( ) [private]
```

Definition at line 303 of file [Raytracing.cpp](#).

```

00303     {
00304     // load in functionality for raytracing shader group handles
00305     PFN_vkGetRayTracingShaderGroupHandlesKHR
00306     pvkGetRayTracingShaderGroupHandlesKHR =
00307         (PFN_vkGetRayTracingShaderGroupHandlesKHR) vkGetDeviceProcAddr(
00308             device->getLogicalDevice(),
00309             "vkGetRayTracingShaderGroupHandlesKHR");
00310
00311     raytracing_properties = VkPhysicalDeviceRayTracingPipelinePropertiesKHR{};
00312     raytracing_properties.sType =
00313         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR;
00314
00315     VkPhysicalDeviceProperties2 properties{};
00316     properties.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;
00317     properties.pNext = &raytracing_properties;
00318
00319     vkGetPhysicalDeviceProperties2(device->getPhysicalDevice(), &properties);
00320
00321     uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
00322     uint32_t handle_size_aligned =
00323         align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);
00324
00325     uint32_t group_count = static_cast<uint32_t>(shader_groups.size());
00326     uint32_t sbt_size = group_count * handle_size_aligned;
00327
00328     std::vector<uint8_t> handles(sbt_size);
00329
00330     VkResult result = pvkGetRayTracingShaderGroupHandlesKHR(
00331         device->getLogicalDevice(), graphicsPipeline, 0, group_count, sbt_size,
00332         handles.data());
00333     ASSERT_VULKAN(result, "Failed to get ray tracing shader group handles!")
00334
00335     const VkBufferUsageFlags bufferUsageFlags =
00336         VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR |
00337         VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT;
00338     const VkMemoryPropertyFlags memoryUsageFlags =
00339         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
00340         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT;
00341
00342     raygenShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
00343                                         memoryUsageFlags);
00344
00345     missShaderBindingTableBuffer.create(device, 2 * handle_size, bufferUsageFlags,
00346                                         memoryUsageFlags);
00347
00348     hitShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
00349                                         memoryUsageFlags);
00350
00351     void* mapped_raygen = nullptr;
00352     vkMapMemory(device->getLogicalDevice(),
00353                 raygenShaderBindingTableBuffer.getBufferMemory(), 0,
00354                 VK_WHOLE_SIZE, 0, &mapped_raygen);
00355
00356     void* mapped_miss = nullptr;
00357     vkMapMemory(device->getLogicalDevice(),
00358                 missShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
  
```

```

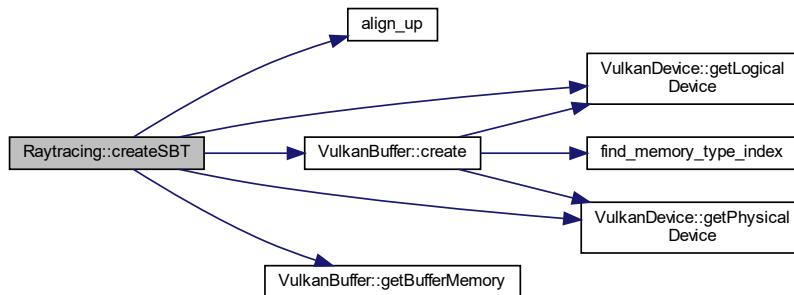
00359         0, &mapped_miss);
00360
00361     void* mapped_rchit = nullptr;
00362     vkMapMemory(device->getLogicalDevice(),
00363                 hitShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
00364                 0, &mapped_rchit);
00365
00366     memcpy(mapped_raygen, handles.data(), handle_size);
00367     memcpy(mapped_miss, handles.data() + handle_size_aligned, handle_size * 2);
00368     memcpy(mapped_rchit, handles.data() + handle_size_aligned * 3, handle_size);
00369 }

```

References [align\\_up\(\)](#), [VulkanBuffer::create\(\)](#), [device](#), [VulkanBuffer::getBufferMemory\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanDevice::getPhysicalDevice\(\)](#), [graphicsPipeline](#), [hitShaderBindingTableBuffer](#), [missShaderBindingTableBuffer](#), [raygenShaderBindingTableBuffer](#), [raytracing\\_properties](#), and [shader\\_groups](#).

Referenced by [init\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.28.3.5 init()

```

void Raytracing::init (
    VulkanDevice * device,
    const std::vector< VkDescriptorSetLayout > & descriptorSetLayouts )

```

Definition at line 14 of file [Raytracing.cpp](#).

```

00016
00017     this->device = device;
00018
00019     createPCRange();
00020     createGraphicsPipeline(descriptorSetLayouts);
00021     createSBT();
}

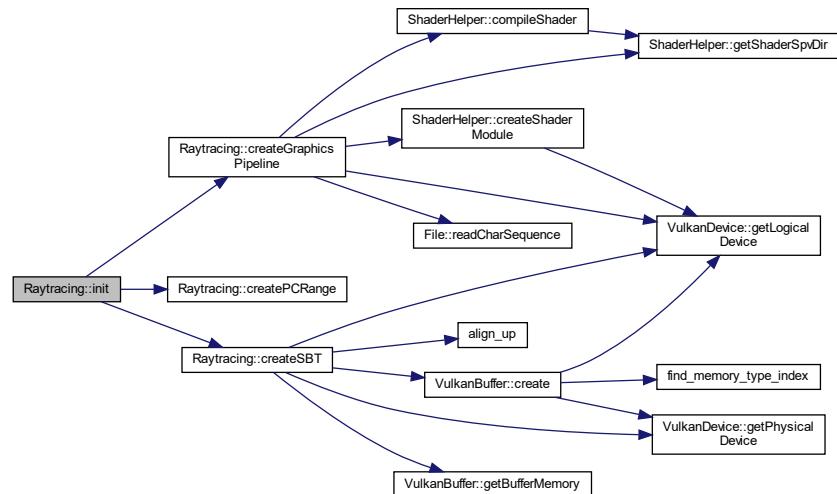
```

```
00022 }
```

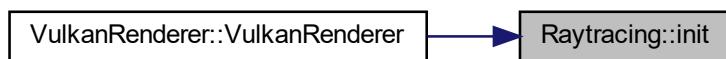
References [createGraphicsPipeline\(\)](#), [createPCRange\(\)](#), [createSBT\(\)](#), and [device](#).

Referenced by [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.28.3.6 recordCommands()

```
void Raytracing::recordCommands (
    VkCommandBuffer & commandBuffer,
    VulkanSwapChain * vulkanSwapChain,
    const std::vector< VkDescriptorSet > & descriptorSets )
```

Definition at line 30 of file [Raytracing.cpp](#).

```
00032
00033     uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
00034     uint32_t handle_size_aligned =
00035         align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);
00036     PFN_vkGetBufferDeviceAddressKHR vkGetBufferDeviceAddressKHR =
00037 
```

```

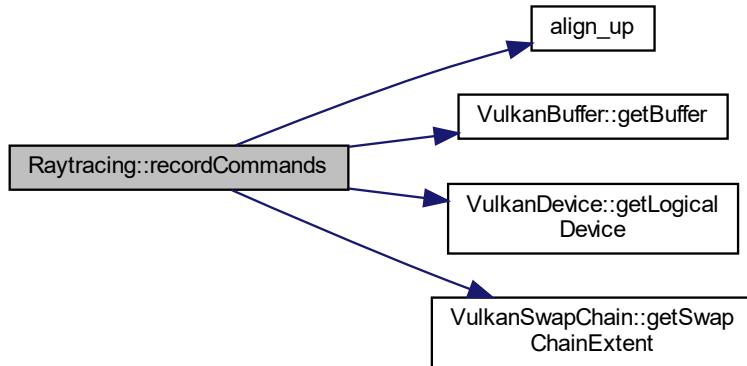
00038     reinterpret_cast<PFN_vkGetBufferDeviceAddressKHR>(vkGetDeviceProcAddr(
00039         device->getLogicalDevice(), "vkGetBufferDeviceAddressKHR"));
00040
00041     PFN_vkCmdTraceRaysKHR pvkCmdTraceRaysKHR =
00042         (PFN_vkCmdTraceRaysKHR)vkGetDeviceProcAddr(device->getLogicalDevice(),
00043             "vkCmdTraceRaysKHR");
00044
00045     VkBufferDeviceAddressInfoKHR bufferDeviceAI{};
00046     bufferDeviceAI.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00047     bufferDeviceAI.buffer = raygenShaderBindingTableBuffer.getBuffer();
00048
00049     rgen_region.deviceAddress =
00050         vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00051     rgen_region.stride = handle_size_aligned;
00052     rgen_region.size = handle_size_aligned;
00053
00054     bufferDeviceAI.buffer = missShaderBindingTableBuffer.getBuffer();
00055     miss_region.deviceAddress =
00056         vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00057     miss_region.stride = handle_size_aligned;
00058     miss_region.size = handle_size_aligned;
00059
00060     bufferDeviceAI.buffer = hitShaderBindingTableBuffer.getBuffer();
00061     hit_region.deviceAddress =
00062         vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00063     hit_region.stride = handle_size_aligned;
00064     hit_region.size = handle_size_aligned;
00065
00066     // for GCC doen't allow references on rvalues go like that ...
00067     pc.clear_color = {0.2f, 0.65f, 0.4f, 1.0f};
00068     // just "Push" constants to given shader stage directly (no buffer)
00069     vkCmdPushConstants(commandBuffer, pipeline_layout,
00070         VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00071             VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00072             VK_SHADER_STAGE_MISS_BIT_KHR,
00073         sizeof(PushConstantRaytracing), &pc);
00074
00075     vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
00076                     graphicsPipeline);
00077
00078     vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
00079         pipeline_layout, 0,
00080             static_cast<uint32_t>(descriptorSets.size()),
00081             descriptorSets.data(), 0, nullptr);
00082
00083     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00084     pvkCmdTraceRaysKHR(commandBuffer, &rgen_region, &miss_region, &hit_region,
00085         &call_region, swap_chain_extent.width,
00086         swap_chain_extent.height, 1);
00087 }

```

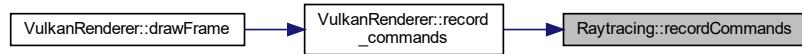
References `align_up()`, `call_region`, `PushConstantRaytracing::clear_color`, `device`, `VulkanBuffer::getBuffer()`, `VulkanDevice::getLogicalDevice()`, `VulkanSwapChain::getSwapChainExtent()`, `graphicsPipeline`, `hit_region`, `hitShaderBindingTableBuffer`, `miss_region`, `missShaderBindingTableBuffer`, `pc`, `pipeline_layout`, `raygenShaderBindingTableBuffer`, `raytracing_properties`, `rgen_region`, and `vulkanSwapChain`.

Referenced by `VulkanRenderer::record_commands()`.

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.28.3.7 `shaderHotReload()`

```
void Raytracing::shaderHotReload (
    const std::vector< VkDescriptorSetLayout > & descriptor_set_layouts )
```

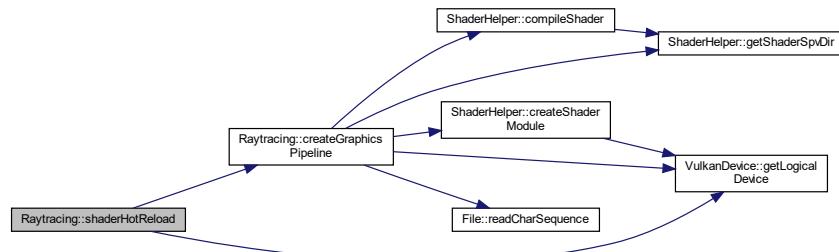
Definition at line 24 of file `Raytracing.cpp`.

```
00025
00026     vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
00027     createGraphicsPipeline(descriptor_set_layouts);
00028 }
```

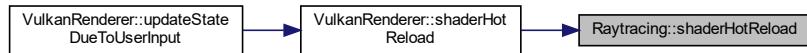
References `createGraphicsPipeline()`, `device`, `VulkanDevice::getLogicalDevice()`, and `graphicsPipeline`.

Referenced by `VulkanRenderer::shaderHotReload()`.

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.28.4 Field Documentation

### 5.28.4.1 call\_region

```
VkStridedDeviceAddressRegionKHR Raytracing::call_region {} [private]
```

Definition at line 45 of file [Raytracing.hpp](#).

Referenced by [recordCommands\(\)](#).

### 5.28.4.2 device

```
VulkanDevice* Raytracing::device {VK_NULL_HANDLE} [private]
```

Definition at line 28 of file [Raytracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createGraphicsPipeline\(\)](#), [createSBT\(\)](#), [init\(\)](#), [recordCommands\(\)](#), and [shaderHotReload\(\)](#).

#### 5.28.4.3 **graphicsPipeline**

```
VkPipeline Raytracing::graphicsPipeline {VK_NULL_HANDLE} [private]
```

Definition at line 31 of file [Raytracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createGraphicsPipeline\(\)](#), [createSBT\(\)](#), [recordCommands\(\)](#), and [shaderHotReload\(\)](#).

#### 5.28.4.4 **hit\_region**

```
VkStridedDeviceAddressRegionKHR Raytracing::hit_region {} [private]
```

Definition at line 44 of file [Raytracing.hpp](#).

Referenced by [recordCommands\(\)](#).

#### 5.28.4.5 **hitShaderBindingTableBuffer**

```
VulkanBuffer Raytracing::hitShaderBindingTableBuffer [private]
```

Definition at line 40 of file [Raytracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createSBT\(\)](#), and [recordCommands\(\)](#).

#### 5.28.4.6 **miss\_region**

```
VkStridedDeviceAddressRegionKHR Raytracing::miss_region {} [private]
```

Definition at line 43 of file [Raytracing.hpp](#).

Referenced by [recordCommands\(\)](#).

#### 5.28.4.7 **missShaderBindingTableBuffer**

```
VulkanBuffer Raytracing::missShaderBindingTableBuffer [private]
```

Definition at line 39 of file [Raytracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createSBT\(\)](#), and [recordCommands\(\)](#).

#### 5.28.4.8 pc

```
PushConstantRaytracing Raytracing::pc {glm::vec4(0.f)} [private]
```

Definition at line 33 of file [Raytracing.hpp](#).

Referenced by [recordCommands\(\)](#).

#### 5.28.4.9 pc\_ranges

```
VkPushConstantRange Raytracing::pc_ranges {VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0, 0} [private]
```

Definition at line 34 of file [Raytracing.hpp](#).

Referenced by [createGraphicsPipeline\(\)](#), and [createPCRange\(\)](#).

#### 5.28.4.10 pipeline\_layout

```
VkPipelineLayout Raytracing::pipeline_layout {VK_NULL_HANDLE} [private]
```

Definition at line 32 of file [Raytracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createGraphicsPipeline\(\)](#), and [recordCommands\(\)](#).

#### 5.28.4.11 raygenShaderBindingTableBuffer

```
VulkanBuffer Raytracing::raygenShaderBindingTableBuffer [private]
```

Definition at line 38 of file [Raytracing.hpp](#).

Referenced by [cleanUp\(\)](#), [createSBT\(\)](#), and [recordCommands\(\)](#).

#### 5.28.4.12 raytracing\_properties

```
VkPhysicalDeviceRayTracingPipelinePropertiesKHR Raytracing::raytracing_properties {} [private]
```

Definition at line 47 of file [Raytracing.hpp](#).

Referenced by [createSBT\(\)](#), and [recordCommands\(\)](#).

#### 5.28.4.13 rgen\_region

```
VkStridedDeviceAddressRegionKHR Raytracing::rgen_region {} [private]
```

Definition at line 42 of file [Raytracing.hpp](#).

Referenced by [recordCommands\(\)](#).

#### 5.28.4.14 shader\_groups

```
std::vector<VkRayTracingShaderGroupCreateInfoKHR> Raytracing::shader_groups [private]
```

Definition at line 36 of file [Raytracing.hpp](#).

Referenced by [createGraphicsPipeline\(\)](#), and [createSBT\(\)](#).

#### 5.28.4.15 shaderBindingTableBuffer

```
VulkanBuffer Raytracing::shaderBindingTableBuffer [private]
```

Definition at line 37 of file [Raytracing.hpp](#).

Referenced by [cleanUp\(\)](#).

#### 5.28.4.16 vulkanSwapChain

```
VulkanSwapChain* Raytracing::vulkanSwapChain {VK_NULL_HANDLE} [private]
```

Definition at line 29 of file [Raytracing.hpp](#).

Referenced by [recordCommands\(\)](#).

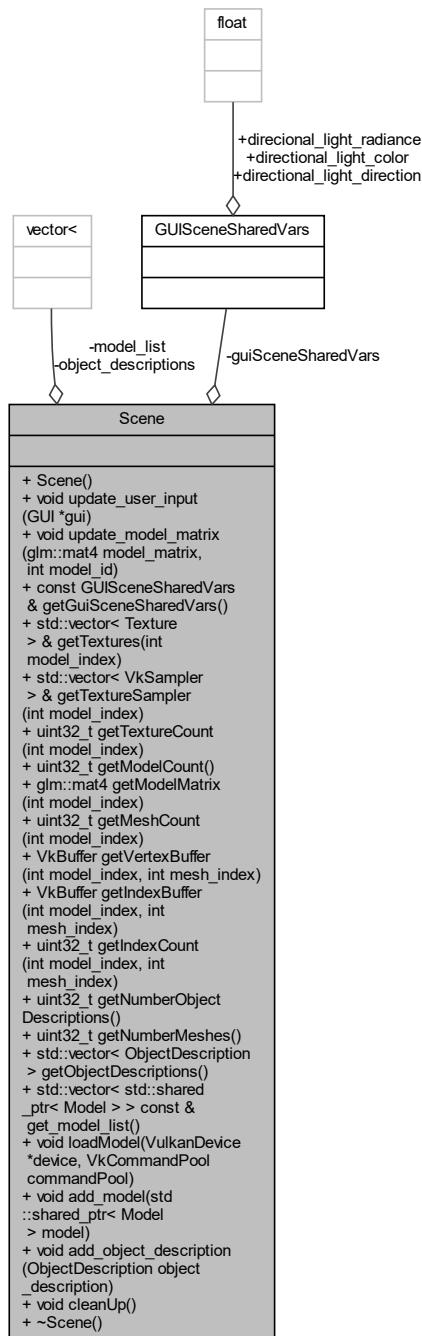
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[Raytracing.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/[Raytracing.cpp](#)

## 5.29 Scene Class Reference

```
#include <Scene.hpp>
```

Collaboration diagram for Scene:



### Public Member Functions

- [Scene \(\)](#)

- void `update_user_input` (GUI \*gui)
- void `update_model_matrix` (glm::mat4 model\_matrix, int model\_id)
- const GUISceneSharedVars & `getGuiSceneSharedVars` ()
- std::vector< Texture > & `getTextures` (int model\_index)
- std::vector< VkSampler > & `getTextureSampler` (int model\_index)
- uint32\_t `getTextureCount` (int model\_index)
- uint32\_t `getModelCount` ()
- glm::mat4 `getModelMatrix` (int model\_index)
- uint32\_t `getMeshCount` (int model\_index)
- VkBuffer `getVertexBuffer` (int model\_index, int mesh\_index)
- VkBuffer `getIndexBuffer` (int model\_index, int mesh\_index)
- uint32\_t `getIndexCount` (int model\_index, int mesh\_index)
- uint32\_t `getNumberObjectDescriptions` ()
- uint32\_t `getNumberMeshes` ()
- std::vector< ObjectDescription > `getObjectDescriptions` ()
- std::vector< std::shared\_ptr< Model > > const & `get_model_list` ()
- void `loadModel` (VulkanDevice \*device, VkCommandPool commandPool)
- void `add_model` (std::shared\_ptr< Model > model)
- void `add_object_description` (ObjectDescription object\_description)
- void `cleanUp` ()
- ~Scene ()

## Private Attributes

- std::vector< ObjectDescription > object\_descriptions
- std::vector< std::shared\_ptr< Model > > model\_list
- GUISceneSharedVars guiSceneSharedVars

### 5.29.1 Detailed Description

Definition at line 30 of file [Scene.hpp](#).

### 5.29.2 Constructor & Destructor Documentation

#### 5.29.2.1 Scene()

```
Scene::Scene ( )
```

Definition at line 3 of file [Scene.cpp](#).  
00003 { }

#### 5.29.2.2 ~Scene()

```
Scene::~Scene ( )
```

Definition at line 55 of file [Scene.cpp](#).  
00055 { }

### 5.29.3 Member Function Documentation

#### 5.29.3.1 add\_model()

```
void Scene::add_model (
    std::shared_ptr< Model > model )
```

Definition at line 22 of file [Scene.cpp](#).

```
00022
00023     model_list.push_back(model);
00024     object_descriptions.push_back(model->getObjectName());
00025 }
```

References [model\\_list](#), and [object\\_descriptions](#).

Referenced by [loadModel\(\)](#).

Here is the caller graph for this function:



#### 5.29.3.2 add\_object\_description()

```
void Scene::add_object_description (
    ObjectDescription object_description )
```

Definition at line 27 of file [Scene.cpp](#).

```
00027
00028     object_descriptions.push_back(object_description);
00029 }
```

References [object\\_descriptions](#).

#### 5.29.3.3 cleanUp()

```
void Scene::cleanUp ( )
```

Definition at line 39 of file [Scene.cpp](#).

```
00039
00040     for (std::shared_ptr<Model> model : model_list) {
00041         model->cleanUp();
00042     }
00043 }
```

References [model\\_list](#).

### 5.29.3.4 get\_model\_list()

```
std::vector< std::shared_ptr< Model > > const & Scene::get_model_list ( ) [inline]
```

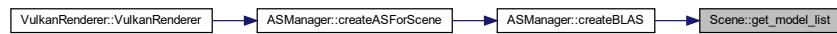
Definition at line 73 of file [Scene.hpp](#).

```
00073
00074     return model_list;
00075 }
```

References [model\\_list](#).

Referenced by [ASManager::createBLAS\(\)](#).

Here is the caller graph for this function:



### 5.29.3.5 getGuiSceneSharedVars()

```
const GUISceneSharedVars & Scene::getGuiSceneSharedVars ( ) [inline]
```

Definition at line 37 of file [Scene.hpp](#).

```
00037
00038     return guiSceneSharedVars;
00039 }
```

References [guiSceneSharedVars](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.29.3.6 getIndexBuffer()

```
VkBuffer Scene::getIndexBuffer (
    int model_index,
    int mesh_index ) [inline]
```

Definition at line 60 of file [Scene.hpp](#).

```
00060
00061     return model_list[model_index]->getMesh(mesh_index)->getIndexBuffer();
00062 }
```

References [model\\_list](#).

Referenced by [Rasterizer::recordCommands\(\)](#).

Here is the caller graph for this function:



### 5.29.3.7 getIndexCount()

```
uint32_t Scene::getIndexCount (
    int model_index,
    int mesh_index ) [inline]
```

Definition at line 63 of file [Scene.hpp](#).

```
00063
00064     return model_list[model_index]->getMesh(mesh_index)->getIndexCount();
00065 }
```

References [model\\_list](#).

Referenced by [Rasterizer::recordCommands\(\)](#).

Here is the caller graph for this function:



### 5.29.3.8 getMeshCount()

```
uint32_t Scene::getMeshCount (
    int model_index ) [inline]
```

Definition at line 54 of file [Scene.hpp](#).

```
00054             {
00055     return static_cast<uint32_t>(model_list[model_index]->getMeshCount ());
00056 }
```

References [model\\_list](#).

Referenced by [Rasterizer::recordCommands\(\)](#).

Here is the caller graph for this function:



### 5.29.3.9 getModelCount()

```
uint32_t Scene::getModelCount () [inline]
```

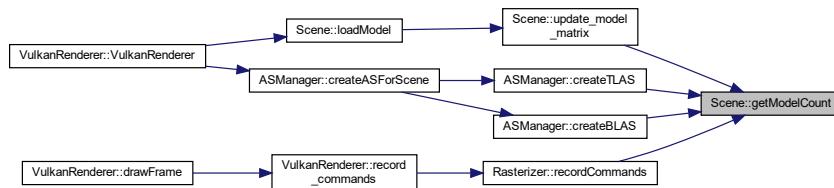
Definition at line 50 of file [Scene.hpp](#).

```
00050 { return static_cast<uint32_t>(model_list.size()); };
```

References [model\\_list](#).

Referenced by [ASManager::createBLAS\(\)](#), [ASManager::createTLAS\(\)](#), [Rasterizer::recordCommands\(\)](#), and [update\\_model\\_matrix\(\)](#).

Here is the caller graph for this function:



### 5.29.3.10 getModelMatrix()

```
glm::mat4 Scene::getModelMatrix (
    int model_index ) [inline]
```

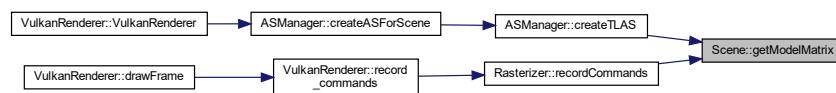
Definition at line 51 of file [Scene.hpp](#).

```
00051     {
00052         return model_list[model_index]->getModel();
00053     };
```

References [model\\_list](#).

Referenced by [ASManager::createTLAS\(\)](#), and [Rasterizer::recordCommands\(\)](#).

Here is the caller graph for this function:



### 5.29.3.11 getNumberMeshes()

```
uint32_t Scene::getNumberMeshes ( )
```

Definition at line 45 of file [Scene.cpp](#).

```
00045     {
00046     uint32_t number_of_meshes = 0;
00047
00048     for (std::shared_ptr<Model> mesh_model : model_list) {
00049         number_of_meshes += static_cast<uint32_t>(mesh_model->getMeshCount ());
00050     }
00051
00052     return number_of_meshes;
00053 }
```

References [model\\_list](#).

### 5.29.3.12 getNumberObjectDescriptions()

```
uint32_t Scene::getNumberObjectDescriptions ( ) [inline]
```

Definition at line 66 of file [Scene.hpp](#).

```
00066     {
00067     return static_cast<uint32_t>(object_descriptions.size());
00068 }
```

References [object\\_descriptions](#).

### 5.29.3.13 getObjectDescriptions()

```
std::vector< ObjectDescription > Scene::getObjectDescriptions () [inline]
```

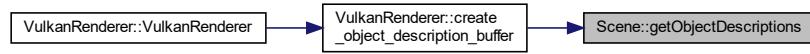
Definition at line 70 of file [Scene.hpp](#).

```
00070
00071     return object_descriptions;
00072 }
```

References [object\\_descriptions](#).

Referenced by [VulkanRenderer::create\\_object\\_description\\_buffer\(\)](#).

Here is the caller graph for this function:



### 5.29.3.14 getTextureCount()

```
uint32_t Scene::getTextureCount (
    int model_index ) [inline]
```

Definition at line 47 of file [Scene.hpp](#).

```
00047
00048     return model_list[model_index]->getTextureCount ();
00049 }
```

References [model\\_list](#).

Referenced by [VulkanRenderer::updateTexturesInSharedRenderDescriptorSet\(\)](#).

Here is the caller graph for this function:



### 5.29.3.15 `getTextures()`

```
std::vector< Texture > & Scene::getTextures (
    int model_index ) [inline]
```

Definition at line 41 of file [Scene.hpp](#).

```
00041
00042     return model_list[model_index]->getTextures();
00043 }
```

References [model\\_list](#).

Referenced by [VulkanRenderer::updateTexturesInSharedRenderDescriptorSet\(\)](#).

Here is the caller graph for this function:



### 5.29.3.16 `getTextureSampler()`

```
std::vector< VkSampler > & Scene::getTextureSampler (
    int model_index ) [inline]
```

Definition at line 44 of file [Scene.hpp](#).

```
00044
00045     return model_list[model_index]->getTextureSamplers();
00046 }
```

References [model\\_list](#).

Referenced by [VulkanRenderer::updateTexturesInSharedRenderDescriptorSet\(\)](#).

Here is the caller graph for this function:



### 5.29.3.17 getVertexBuffer()

```
VkBuffer Scene::getVertexBuffer (
    int model_index,
    int mesh_index ) [inline]
```

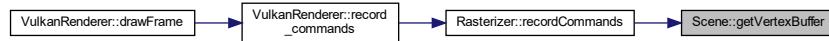
Definition at line 57 of file [Scene.hpp](#).

```
00057
00058     return model_list[model_index]->getMesh(mesh_index)->getVertexBuffer();
00059 }
```

References [model\\_list](#).

Referenced by [Rasterizer::recordCommands\(\)](#).

Here is the caller graph for this function:



### 5.29.3.18 loadModel()

```
void Scene::loadModel (
    VulkanDevice * device,
    VkCommandPool commandPool )
```

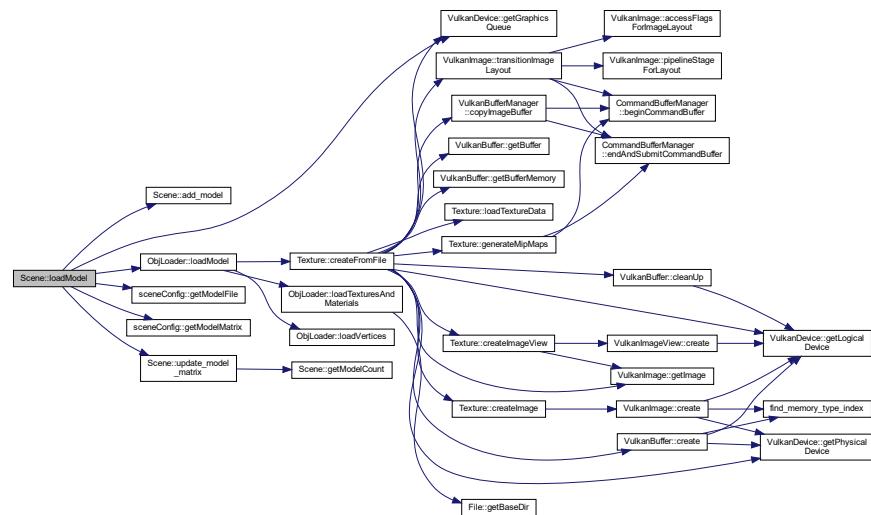
Definition at line 9 of file [Scene.cpp](#).

```
00009
00010 ObjLoader obj_loader(device, device->getGraphicsQueue(), commandPool);
00011
00012 std::string modelName = sceneConfig::getModelFile();
00013 std::shared_ptr<Model> new_model = obj_loader.loadModel(modelName);
00014 add_model(new_model);
00015
00016 glm::mat4 modelMatrix = sceneConfig::getModelMatrix();
00017 update_model_matrix(modelMatrix, 0);
00018
00019
00020 }
```

References [add\\_model\(\)](#), [VulkanDevice::getGraphicsQueue\(\)](#), [sceneConfig::getModelFile\(\)](#), [sceneConfig::getModelMatrix\(\)](#), [ObjLoader::loadModel\(\)](#), and [update\\_model\\_matrix\(\)](#).

Referenced by [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### **5.29.3.19 update\_model\_matrix()**

```
void Scene::update_model_matrix (
```

Definition at line 31 of file [Scene.cpp](#).

```
00031                                     {  
00032     if (model_id >= static_cast<int32_t>(getModelCount()) || model_id < 0) {  
00033         throw std::runtime_error("Wrong model id value!");  
00034     }  
00035  
00036     model_list[model_id]->set_model(model_matrix);  
00037 }
```

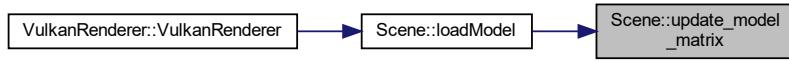
References `getModelCount()`, and `model_list`.

Referenced by [loadModel\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.29.3.20 update\_user\_input()

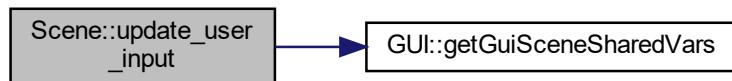
```
void Scene::update_user_input (
    GUI * gui )
```

Definition at line 5 of file [Scene.cpp](#).

```
00005 {  
00006     guiSceneSharedVars = gui->getGuiSceneSharedVars();  
00007 }
```

References [GUI::getGuiSceneSharedVars\(\)](#), and [guiSceneSharedVars](#).

Here is the call graph for this function:



## 5.29.4 Field Documentation

#### 5.29.4.1 guiSceneSharedVars

```
GUISceneSharedVars Scene::guiSceneSharedVars [private]
```

Definition at line 89 of file [Scene.hpp](#).

Referenced by [getGuiSceneSharedVars\(\)](#), and [update\\_user\\_input\(\)](#).

#### 5.29.4.2 model\_list

```
std::vector<std::shared_ptr<Model>> Scene::model_list [private]
```

Definition at line 87 of file [Scene.hpp](#).

Referenced by [add\\_model\(\)](#), [cleanUp\(\)](#), [get\\_model\\_list\(\)](#), [getIndexBuffer\(\)](#), [getIndexCount\(\)](#), [getMeshCount\(\)](#), [getModelCount\(\)](#), [getModelMatrix\(\)](#), [getNumberMeshes\(\)](#), [getTextureCount\(\)](#), [getTextures\(\)](#), [getTextureSampler\(\)](#), [getVertexBuffer\(\)](#), and [update\\_model\\_matrix\(\)](#).

#### 5.29.4.3 object\_descriptions

```
std::vector<ObjectDescription> Scene::object_descriptions [private]
```

Definition at line 86 of file [Scene.hpp](#).

Referenced by [add\\_model\(\)](#), [add\\_object\\_description\(\)](#), [getNumberObjectDescriptions\(\)](#), and [getObjectDescriptions\(\)](#).

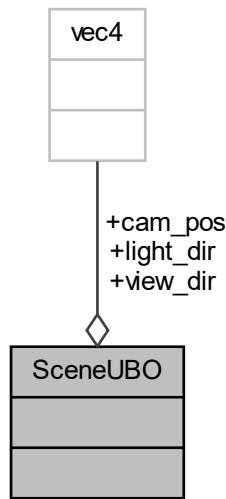
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[Scene.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/[Scene.cpp](#)

## 5.30 SceneUBO Struct Reference

```
#include <SceneUBO.hpp>
```

Collaboration diagram for SceneUBO:



## Data Fields

- `vec4 light_dir`
- `vec4 view_dir`
- `vec4 cam_pos`

### 5.30.1 Detailed Description

Definition at line 17 of file [SceneUBO.hpp](#).

### 5.30.2 Field Documentation

#### 5.30.2.1 cam\_pos

`vec4 SceneUBO::cam_pos`

Definition at line 21 of file [SceneUBO.hpp](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

### 5.30.2.2 light\_dir

```
vec4 SceneUBO::light_dir
```

Definition at line 18 of file [SceneUBO.hpp](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

### 5.30.2.3 view\_dir

```
vec4 SceneUBO::view_dir
```

Definition at line 19 of file [SceneUBO.hpp](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

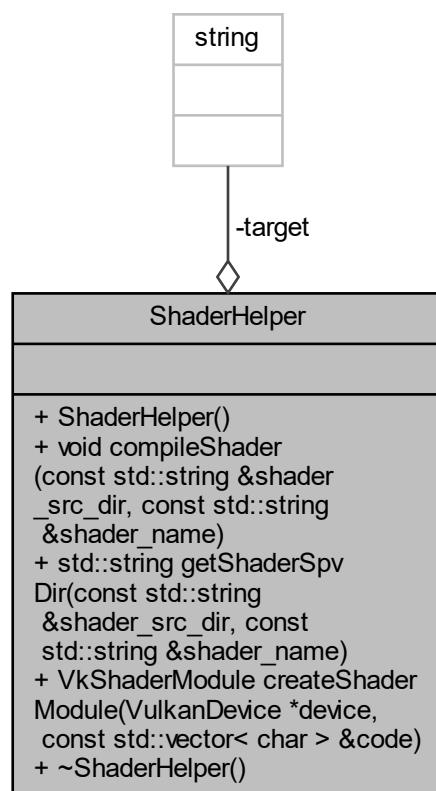
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[SceneUBO.hpp](#)

## 5.31 ShaderHelper Class Reference

```
#include <ShaderHelper.hpp>
```

Collaboration diagram for ShaderHelper:



## Public Member Functions

- `ShaderHelper ()`
- `void compileShader (const std::string &shader_src_dir, const std::string &shader_name)`
- `std::string getShaderSpvDir (const std::string &shader_src_dir, const std::string &shader_name)`
- `VkShaderModule createShaderModule (VulkanDevice *device, const std::vector< char > &code)`
- `~ShaderHelper ()`

## Private Attributes

- `std::string target = " --target-env=vulkan1.3 "`

### 5.31.1 Detailed Description

Definition at line 9 of file [ShaderHelper.hpp](#).

### 5.31.2 Constructor & Destructor Documentation

#### 5.31.2.1 `ShaderHelper()`

```
ShaderHelper::ShaderHelper ( )
```

Definition at line 8 of file [ShaderHelper.cpp](#).  
00008 { }

#### 5.31.2.2 `~ShaderHelper()`

```
ShaderHelper::~ShaderHelper ( )
```

Definition at line 67 of file [ShaderHelper.cpp](#).  
00067 { }

### 5.31.3 Member Function Documentation

### 5.31.3.1 compileShader()

```
void ShaderHelper::compileShader (
    const std::string & shader_src_dir,
    const std::string & shader_name )
```

Definition at line 10 of file [ShaderHelper.cpp](#).

```
00011 {  
00012     // GLSLC_EXE is set by cmake to the location of the vulkan glslc  
00013     std::stringstream shader_src_path;  
00014     std::stringstream shader_log_file;  
00015     std::stringstream cmdShaderCompile;  
00016     std::stringstream adminPrivileges;  
00017     adminPrivileges << "runas /user:<admin-user> \"\"";  
00018  
00019     // with wrapping your path with quotation marks one can use paths with blanks ...  
00020     shader_src_path << shader_src_dir << shader_name;  
00021     std::string shader_spv_path = getShaderSpvDir(shader_src_dir, shader_name);  
00022     shader_log_file << shader_src_dir << shader_name << ".log.txt";  
00023     std::stringstream log_stdout_and_stderr;  
00024     log_stdout_and_stderr << " > " << shader_log_file.str() << " 2> "  
00025     << shader_log_file.str();  
00026  
00027     cmdShaderCompile //<< adminPrivileges.str()  
00028     << GLSLC_EXE << target << std::quoted(shader_src_path.str()) << " -o "  
00029     << std::quoted(shader_spv_path);  
00030     //<< log_stdout_and_stderr.str();  
00031  
00032     // std::cout << cmdShaderCompile.str().c_str();  
00033  
00034     system(cmdShaderCompile.str().c_str());  
00035  
00036 }
```

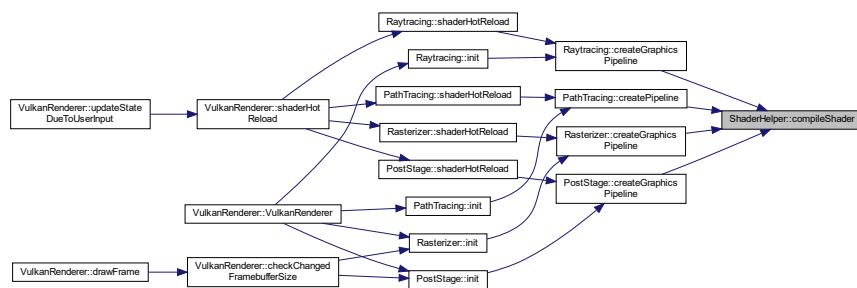
References [getShaderSpvDir\(\)](#), and [target](#).

Referenced by [PostStage::createGraphicsPipeline\(\)](#), [Rasterizer::createGraphicsPipeline\(\)](#), [Raytracing::createGraphicsPipeline\(\)](#), and [PathTracing::createPipeline\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.31.3.2 createShaderModule()

```
VkShaderModule ShaderHelper::createShaderModule (
    VulkanDevice * device,
    const std::vector< char > & code )
```

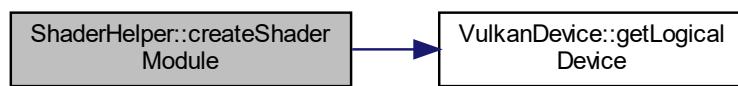
Definition at line 48 of file [ShaderHelper.cpp](#).

```
00049
00050 // shader module create info
00051 VkShaderModuleCreateInfo shader_module_create_info{};
00052 shader_module_create_info.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
00053 shader_module_create_info.codeSize = code.size(); // size of code
00054 shader_module_create_info.pCode =
00055     reinterpret_cast<const uint32_t*>(code.data()); // pointer to code
00056
00057 VulkanDevice shader_module;
00058 VkResult result =
00059     vkCreateShaderModule(device->getLogicalDevice(),
00060                           &shader_module_create_info, nullptr, &shader_module);
00061
00062 ASSERT_VULKAN(result, "Failed to create a shader module!")
00063
00064 return shader_module;
00065 }
```

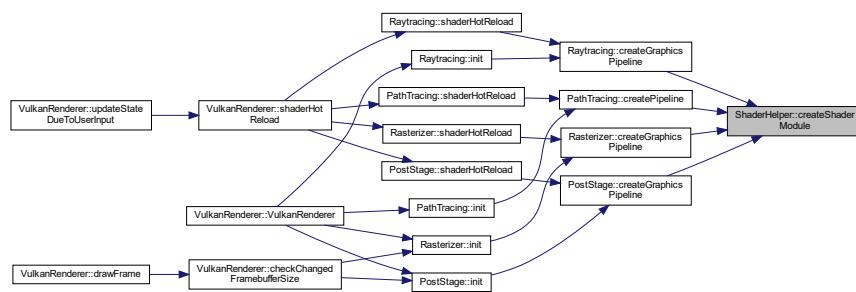
References [VulkanDevice::getLogicalDevice\(\)](#).

Referenced by [PostStage::createGraphicsPipeline\(\)](#), [Rasterizer::createGraphicsPipeline\(\)](#), [Raytracing::createGraphicsPipeline\(\)](#), and [PathTracing::createPipeline\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.31.3.3 getShaderSpvDir()

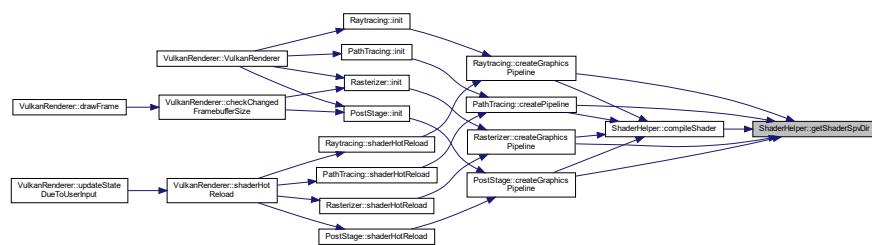
```
std::string ShaderHelper::getShaderSpvDir (
    const std::string & shader_src_dir,
    const std::string & shader_name )
```

Definition at line 38 of file [ShaderHelper.cpp](#).

```
00039
00040     std::string shader_spv_dir = "spv/";
00041
00042     std::stringstream vertShaderSpv;
00043     vertShaderSpv << shader_src_dir << shader_spv_dir << shader_name << ".spv";
00044
00045     return vertShaderSpv.str();
00046 }
```

Referenced by [compileShader\(\)](#), [PostStage::createGraphicsPipeline\(\)](#), [Rasterizer::createGraphicsPipeline\(\)](#), [Raytracing::createGraphicsPipeline\(\)](#), and [PathTracing::createPipeline\(\)](#).

Here is the caller graph for this function:



### 5.31.4 Field Documentation

#### 5.31.4.1 target

```
std::string ShaderHelper::target = " --target-env=vulkan1.3" [private]
```

Definition at line 24 of file [ShaderHelper.hpp](#).

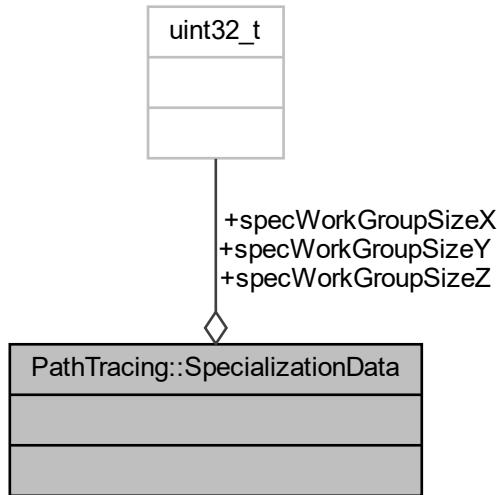
Referenced by [compileShader\(\)](#).

The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/[ShaderHelper.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/[ShaderHelper.cpp](#)

## 5.32 PathTracing::SpecializationData Struct Reference

Collaboration diagram for PathTracing::SpecializationData:



### Data Fields

- `uint32_t specWorkGroupSizeX = 16`
- `uint32_t specWorkGroupSizeY = 8`
- `uint32_t specWorkGroupSizeZ = 0`

#### 5.32.1 Detailed Description

Definition at line 53 of file [PathTracing.hpp](#).

#### 5.32.2 Field Documentation

##### 5.32.2.1 specWorkGroupSizeX

```
uint32_t PathTracing::SpecializationData::specWorkGroupSizeX = 16
```

Definition at line 55 of file [PathTracing.hpp](#).

Referenced by [PathTracing::createPipeline\(\)](#), and [PathTracing::recordCommands\(\)](#).

### 5.32.2.2 specWorkGroupSizeY

```
uint32_t PathTracing::SpecializationData::specWorkGroupSizeY = 8
```

Definition at line 56 of file [PathTracing.hpp](#).

Referenced by [PathTracing::createPipeline\(\)](#), and [PathTracing::recordCommands\(\)](#).

### 5.32.2.3 specWorkGroupSizeZ

```
uint32_t PathTracing::SpecializationData::specWorkGroupSizeZ = 0
```

Definition at line 57 of file [PathTracing.hpp](#).

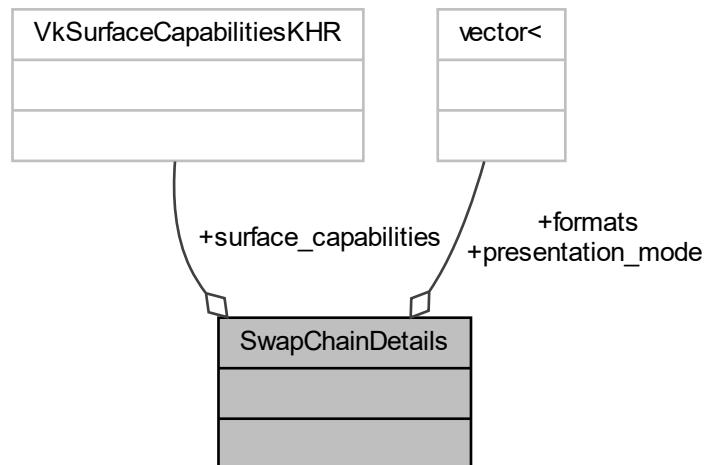
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[PathTracing.hpp](#)

## 5.33 SwapChainDetails Struct Reference

```
#include <SwapChainDetails.hpp>
```

Collaboration diagram for SwapChainDetails:



### Data Fields

- `VkSurfaceCapabilitiesKHR surface_capabilities`
- `std::vector<VkSurfaceFormatKHR> formats`
- `std::vector<VkPresentModeKHR> presentation_mode`

### 5.33.1 Detailed Description

Definition at line 6 of file [SwapChainDetails.hpp](#).

### 5.33.2 Field Documentation

#### 5.33.2.1 formats

```
std::vector<VkSurfaceFormatKHR> SwapChainDetails::formats
```

Definition at line 10 of file [SwapChainDetails.hpp](#).

Referenced by [VulkanDevice::check\\_device\\_suitable\(\)](#), and [VulkanSwapChain::initVulkanContext\(\)](#).

#### 5.33.2.2 presentation\_mode

```
std::vector<VkPresentModeKHR> SwapChainDetails::presentation_mode
```

Definition at line 12 of file [SwapChainDetails.hpp](#).

Referenced by [VulkanDevice::check\\_device\\_suitable\(\)](#), and [VulkanSwapChain::initVulkanContext\(\)](#).

#### 5.33.2.3 surface\_capabilities

```
VkSurfaceCapabilitiesKHR SwapChainDetails::surface_capabilities
```

Definition at line 8 of file [SwapChainDetails.hpp](#).

Referenced by [VulkanSwapChain::initVulkanContext\(\)](#).

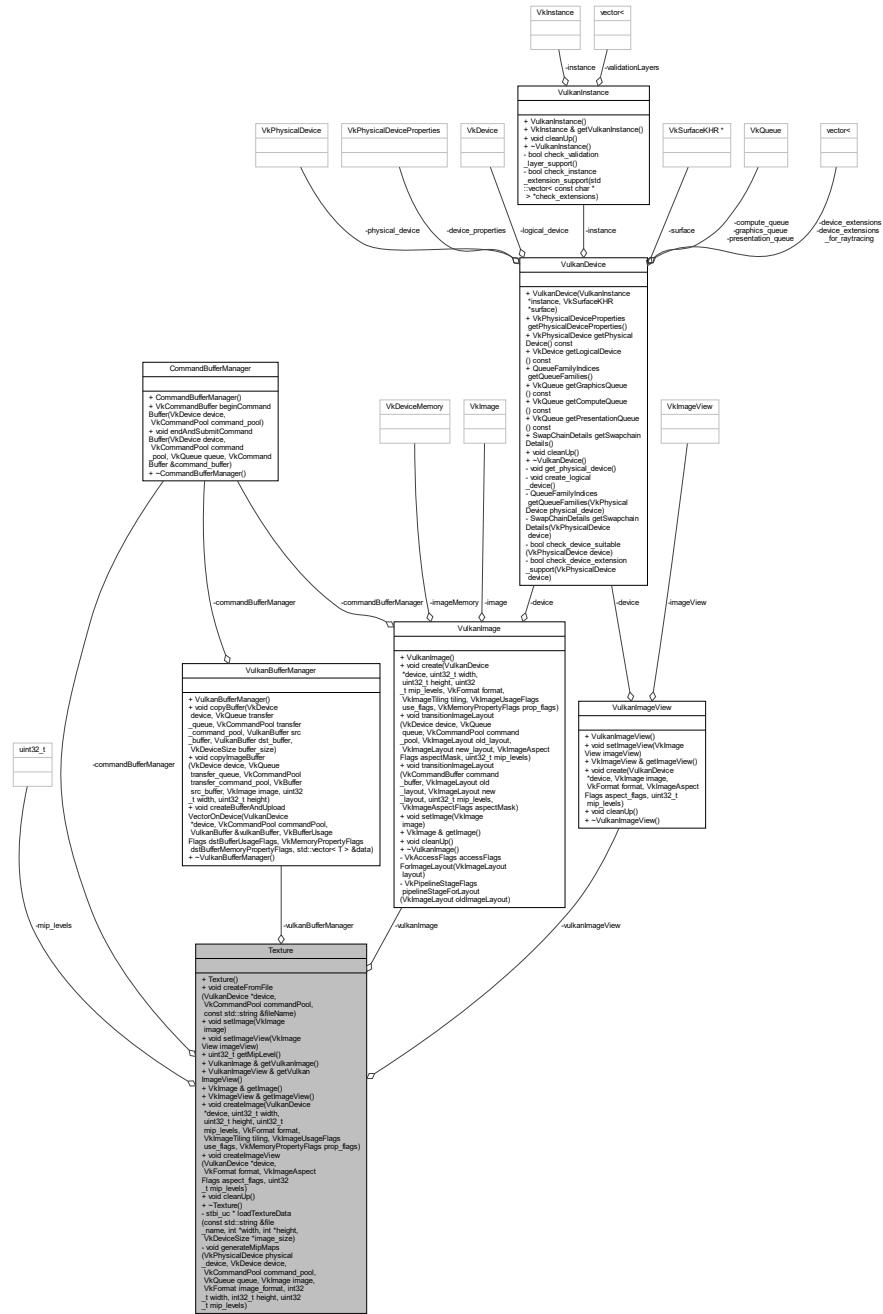
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[SwapChainDetails.hpp](#)

## 5.34 Texture Class Reference

```
#include <Texture.hpp>
```

## Collaboration diagram for Texture:



## Public Member Functions

- `Texture()`
  - `void createFromFile (VulkanDevice *device, VkCommandPool commandPool, const std::string &fileName)`
  - `void setImage (VklImage image)`

- void [setImageView](#) (VkImageView imageView)
- uint32\_t [getMipLevel](#) ()
- [VulkanImage & getVulkanImage](#) ()
- [VulkanImageView & getVulkanImageView](#) ()
- VulkanImage & [getImage](#) ()
- VulkanImageView & [getImageView](#) ()
- void [createImage](#) (VulkanDevice \*device, uint32\_t width, uint32\_t height, uint32\_t [mip\\_levels](#), VkFormat format, VkImageTiling tiling, VkImageUsageFlags use\_flags, VkMemoryPropertyFlags prop\_flags)
- void [createImageView](#) (VulkanDevice \*device, VkFormat format, VkImageAspectFlags aspect\_flags, uint32\_t [mip\\_levels](#))
- void [cleanUp](#) ()
- ~[Texture](#) ()

## Private Member Functions

- stbi\_uc \* [loadTextureData](#) (const std::string &file\_name, int \*width, int \*height, VkDeviceSize \*image\_size)
- void [generateMipMaps](#) (VkPhysicalDevice physical\_device, VkDevice device, VkCommandPool command\_pool, VkQueue queue, VulkanImage image, VkFormat image\_format, int32\_t width, int32\_t height, uint32\_t [mip\\_levels](#))

## Private Attributes

- uint32\_t [mip\\_levels](#) = 0
- CommandBufferManager [commandBufferManager](#)
- VulkanBufferManager [vulkanBufferManager](#)
- VulkanImage [vulkanImage](#)
- VulkanImageView [vulkanImageView](#)

### 5.34.1 Detailed Description

Definition at line 13 of file [Texture.hpp](#).

### 5.34.2 Constructor & Destructor Documentation

#### 5.34.2.1 Texture()

```
Texture::Texture ( )
```

Definition at line 6 of file [Texture.cpp](#).  
00006 { }

### 5.34.2.2 ~Texture()

```
Texture::~Texture ( )
```

Definition at line 89 of file [Texture.cpp](#).  
 00089 {}

## 5.34.3 Member Function Documentation

### 5.34.3.1 cleanUp()

```
void Texture::cleanUp ( )
```

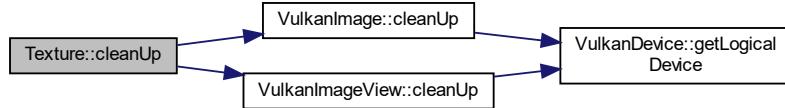
Definition at line 84 of file [Texture.cpp](#).

```
00084      {
00085     vulkanImageView.cleanUp ();
00086     vulkanImage.cleanUp ();
00087 }
```

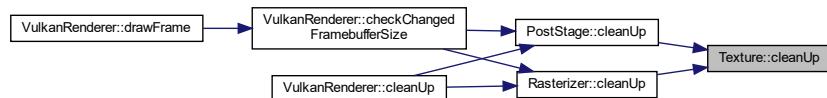
References [VulkanImage::cleanUp\(\)](#), [VulkanImageView::cleanUp\(\)](#), [vulkanImage](#), and [vulkanImageView](#).

Referenced by [PostStage::cleanUp\(\)](#), and [Rasterizer::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.34.3.2 createFromFile()

```
void Texture::createFromFile (
    VulkanDevice * device,
    VkCommandPool commandPool,
    const std::string & fileName )
```

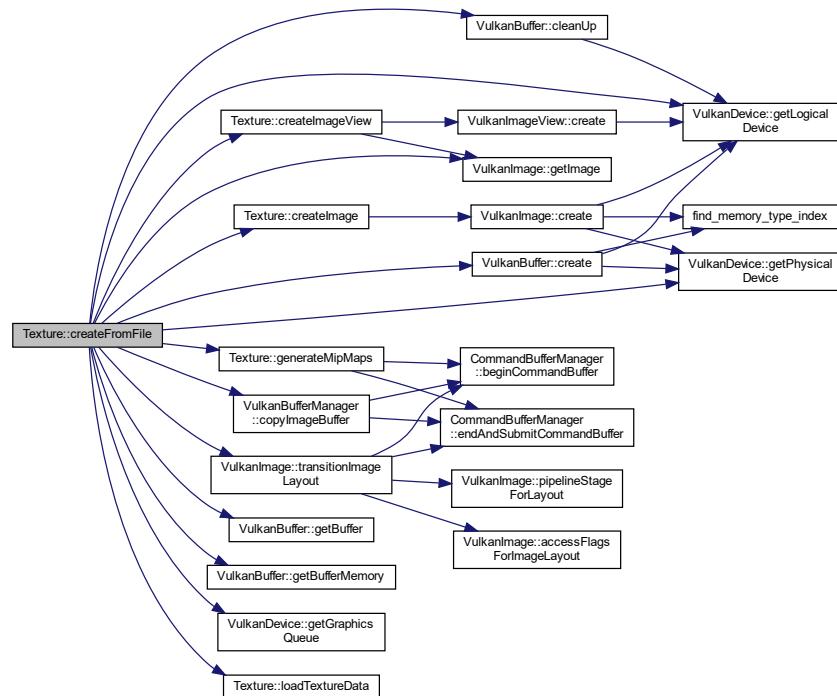
Definition at line 8 of file [Texture.cpp](#).

```
00009
00010     int width, height;
00011     VkDeviceSize size;
00012     stbi_uc* image_data = loadTextureData\(fileName, &width, &height, &size\);
00013
00014     mip_levels =
00015         static_cast<uint32_t>(std::floor(std::log2(std::max(width, height)))) + 1;
00016
00017     // create staging buffer to hold loaded data, ready to copy to device
00018     VulkanBuffer stagingBuffer;
00019     stagingBuffer.create(device, size, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
00020                           VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
00021                           VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);
00022
00023     // copy image data to staging buffer
00024     void* data;
00025     vkMapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory(), 0,
00026                  size, 0, &data);
00027     memcpy(data, image_data, static_cast<size_t>(size));
00028     vkUnmapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory());
00029
00030     // free original image data
00031     stbi_image_free(image_data);
00032
00033     createImage(device, width, height, mip_levels, VK_FORMAT_R8G8B8A8_UNORM,
00034                 VK_IMAGE_TILING_OPTIMAL,
00035                 VK_IMAGE_USAGE_TRANSFER_SRC_BIT |
00036                 VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
00037                 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00038
00039     // copy data to image
00040     // transition image to be DST for copy operation
00041     vulkanImage.transitionImageLayout(
00042         device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
00043         VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
00044         VK_IMAGE_ASPECT_COLOR_BIT, mip_levels);
00045
00046     // copy data to image
00047     vulkanBufferManager.copyImageBuffer(
00048         device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
00049         stagingBuffer.getBuffer(), vulkanImage.getImage(), width, height);
00050
00051     // generate mipmaps
00052     generateMipMaps(device->getPhysicalDevice(), device->getLogicalDevice(),
00053                     commandPool, device->getGraphicsQueue(),
00054                     vulkanImage.getImage(), VK_FORMAT_R8G8B8A8_SRGB, width,
00055                     height, mip_levels);
00056
00057     stagingBuffer.cleanUp();
00058
00059     createImageView(device, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_COLOR_BIT,
00060                      mip_levels);
00061 }
```

References [VulkanBuffer::cleanUp\(\)](#), [VulkanBufferManager::copyImageBuffer\(\)](#), [VulkanBuffer::create\(\)](#), [createImage\(\)](#), [createImageView\(\)](#), [generateMipMaps\(\)](#), [VulkanBuffer::getBuffer\(\)](#), [VulkanBuffer::getBufferMemory\(\)](#), [VulkanDevice::getGraphicsQueue\(\)](#), [VulkanImage::getImage\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanDevice::getPhysicalDevice\(\)](#), [loadTextureData\(\)](#), [mip\\_levels](#), [VulkanImage::transitionImageLayout\(\)](#), [vulkanBufferManager](#), and [vulkanImage](#).

Referenced by [ObjLoader::loadModel\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.34.3.3 createImage()

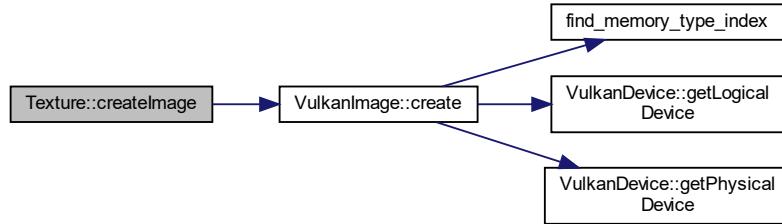
```
void Texture::createImage (
    VulkanDevice * device,
    uint32_t width,
    uint32_t height,
    uint32_t mip_levels,
    VkFormat format,
    VkImageTiling tiling,
    VkImageUsageFlags use_flags,
    VkMemoryPropertyFlags prop_flags )
```

Definition at line 69 of file [Texture.cpp](#).

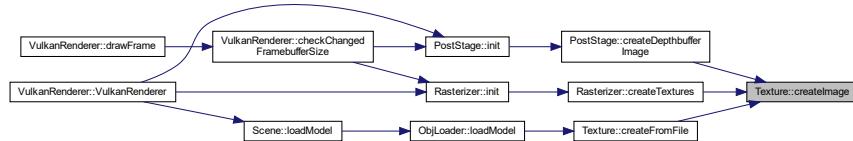
References [VulkanImage::create\(\)](#), [mip\\_levels](#), and [vulkanImage](#).

Referenced by [PostStage::createDepthbufferImage\(\)](#), [createFromFile\(\)](#), and [Rasterizer::createTextures\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.34.3.4 createImageView()

```

void Texture::createImageView (
    VulkanDevice * device,
    VkFormat format,
    VkImageAspectFlags aspect_flags,
    uint32_t mip_levels )
  
```

Definition at line 77 of file [Texture.cpp](#).

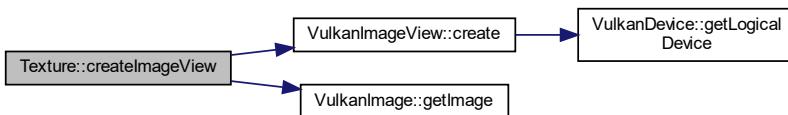
```

00079
00080     vulkanImageView.create(device, vulkanImage.getImage(), format, aspect_flags,
00081                               mip_levels);
00082 }
  
```

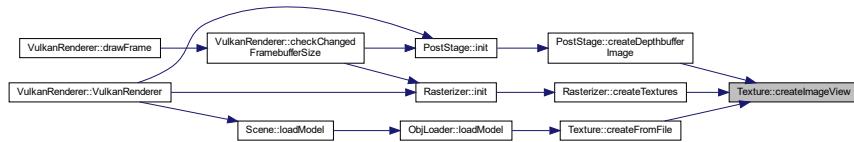
References [VulkanImageView::create\(\)](#), [VulkanImage::getImage\(\)](#), [mip\\_levels](#), [vulkanImage](#), and [vulkanImageView](#).

Referenced by [PostStage::createDepthbufferImage\(\)](#), [createFromFile\(\)](#), and [Rasterizer::createTextures\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.34.3.5 generateMipMaps()

```
void Texture::generateMipMaps (
    VkPhysicalDevice physical_device,
    VkDevice device,
    VkCommandPool command_pool,
    VkQueue queue,
    VkImage image,
    VkFormat image_format,
    int32_t width,
    int32_t height,
    uint32_t mip_levels ) [private]
```

Definition at line 111 of file [Texture.cpp](#).

```
00115 // Check if image format supports linear blitting
00116 VkFormatProperties formatProperties;
00117 vkGetPhysicalDeviceFormatProperties(physical_device, image_format,
00118                                     &formatProperties);
00119
00120 if (!(formatProperties.optimalTilingFeatures &
00121       VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT)) {
00122     throw std::runtime_error(
00123         "Texture image format does not support linear blitting!");
00124 }
00125
00126 VkCommandBuffer command_buffer =
00127     commandBufferManager.beginCommandBuffer(device, command_pool);
00128
00129 VkImageMemoryBarrier barrier{};
00130 barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00131 barrier.image = image;
00132 barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
00133 barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
00134 barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00135 barrier.subresourceRange.baseArrayLayer = 0;
00136 barrier.subresourceRange.layerCount = 1;
00137 barrier.subresourceRange.levelCount = 1;
00138
00139 // TEMP VARS needed for decreasing step by step for factor 2
00140 int32_t tmp_width = width;
00141 int32_t tmp_height = height;
00142
00143 // -- WE START AT 1 !
00144 for (uint32_t i = 1; i < mip_levels; i++) {
00145     // WAIT for previous mip map level for being ready
00146     barrier.subresourceRange.baseMipLevel = i - 1;
00147     // HERE we TRANSITION for having a SRC format now
00148     barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
00149     barrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
00150     barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00151     barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
00152
00153     vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00154                           VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 0,
00155                           nullptr, 1, &barrier);
00156 }
```

```

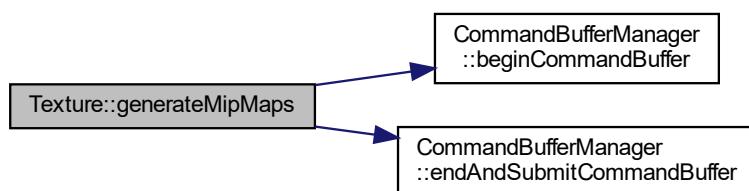
00158 // when barrier over we can now blit :)
00159 VkImageBlit blit{};
00160
00161 // -- OFFSETS describing the 3D-dimesnion of the region
00162 blit.srcOffsets[0] = {0, 0, 0};
00163 blit.srcOffsets[1] = {tmp_width, tmp_height, 1};
00164 blit.srcSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00165 // copy from previous level
00166 blit.srcSubresource.mipLevel = i - 1;
00167 blit.srcSubresource.baseArrayLayer = 0;
00168 blit.srcSubresource.layerCount = 1;
00169 // -- OFFSETS describing the 3D-dimesnion of the region
00170 blit.dstOffsets[0] = {0, 0, 0};
00171 blit.dstOffsets[1] = {tmp_width > 1 ? tmp_width / 2 : 1,
00172                         tmp_height > 1 ? tmp_height / 2 : 1, 1};
00173 blit.dstSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00174 // -- COPY to next mipmap level
00175 blit.dstSubresource.mipLevel = i;
00176 blit.dstSubresource.baseArrayLayer = 0;
00177 blit.dstSubresource.layerCount = 1;
00178
00179 vkCmdBlitImage(command_buffer, image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
00180                  image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &blit,
00181                  VK_FILTER_LINEAR);
00182
00183 // REARRANGE image formats for having the correct image formats again
00184 barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
00185 barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00186 barrier.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
00187 barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
00188
00189 vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00190                       VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, nullptr,
00191                       0, nullptr, 1, &barrier);
00192
00193 if (tmp_width > 1) tmp_width /= 2;
00194 if (tmp_height > 1) tmp_height /= 2;
00195 }
00196
00197 barrier.subresourceRange.baseMipLevel = mip_levels - 1;
00198 barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
00199 barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00200 barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00201 barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
00202
00203 vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00204                       VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, nullptr, 0,
00205                       nullptr, 1, &barrier);
00206
00207 commandBufferManager.endAndSubmitCommandBuffer(device, command_pool, queue,
00208                                                 command_buffer);
00209 }

```

References [CommandBufferManager::beginCommandBuffer\(\)](#), [commandBufferManager](#), [CommandBufferManager::endAndSubmitCommandBuffer\(\)](#) and [mip\\_levels](#).

Referenced by [createFromFile\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.34.3.6 getImage()

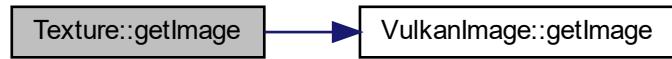
`VkImage & Texture::getImage () [inline]`

Definition at line 26 of file [Texture.hpp](#).

```
00026 { return vulkanImage.getImage(); };
```

References [VulkanImage::getImage\(\)](#), and [vulkanImage](#).

Here is the call graph for this function:



### 5.34.3.7 getImageView()

`VkImageView & Texture::getImageView () [inline]`

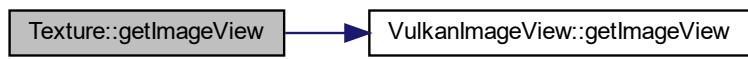
Definition at line 27 of file [Texture.hpp](#).

```
00027 { return vulkanImageView.getImageView(); };
```

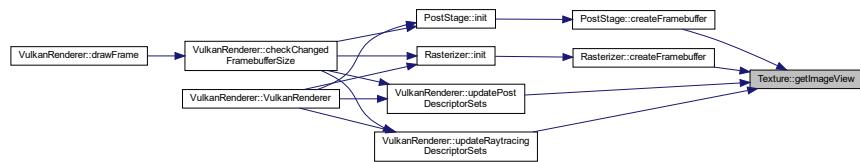
References [VulkanImageView::getImageView\(\)](#), and [vulkanImageView](#).

Referenced by [PostStage::createFramebuffer\(\)](#), [Rasterizer::createFramebuffer\(\)](#), [VulkanRenderer::updatePostDescriptorSets\(\)](#), and [VulkanRenderer::updateRaytracingDescriptorSets\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.34.3.8 getMipLevel()

`uint32_t Texture::getMipLevel ( ) [inline]`

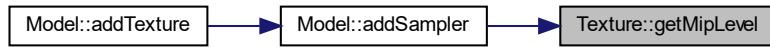
Definition at line 23 of file [Texture.hpp](#).

```
00023 { return mip_levels; };
```

References [mip\\_levels](#).

Referenced by [Model::addSampler\(\)](#).

Here is the caller graph for this function:



### 5.34.3.9 getVulkanImage()

`VulkanImage & Texture::getVulkanImage ( ) [inline]`

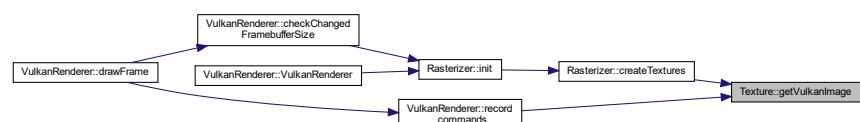
Definition at line 24 of file [Texture.hpp](#).

```
00024 { return vulkanImage; };
```

References [vulkanImage](#).

Referenced by [Rasterizer::createTextures\(\)](#), and [VulkanRenderer::record\\_commands\(\)](#).

Here is the caller graph for this function:



### 5.34.3.10 getVulkanImageView()

```
VulkanImageView & Texture::getVulkanImageView () [inline]
```

Definition at line 25 of file [Texture.hpp](#).

```
00025 { return vulkanImageView; };
```

References [vulkanImageView](#).

### 5.34.3.11 loadTextureData()

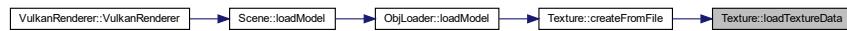
```
stbi_uc * Texture::loadTextureData (
    const std::string & file_name,
    int * width,
    int * height,
    VkDeviceSize * image_size ) [private]
```

Definition at line 91 of file [Texture.cpp](#).

```
00092 {
00093     // number of channels image uses
00094     int channels;
00095     // load pixel data for image
00096     // std::string file_loc = "../Resources/Textures/" + file_name;
00097     stbi_uc* image =
00098         stbi_load(file_name.c_str(), width, height, &channels, STBI_rgb_alpha);
00099
00100     if (!image) {
00101         throw std::runtime_error("Failed to load a texture file! (" + file_name +
00102                                 ")");
00103     }
00104
00105     // calculate image size using given and known data
00106     *image_size = *width * *height * 4;
00107
00108     return image;
00109 }
```

Referenced by [createFromFile\(\)](#).

Here is the caller graph for this function:



### 5.34.3.12 setImage()

```
void Texture::setImage (
    VkImage image )
```

Definition at line 63 of file [Texture.cpp](#).  
 00063 { [vulkanImage.setImage\(image\);](#) }

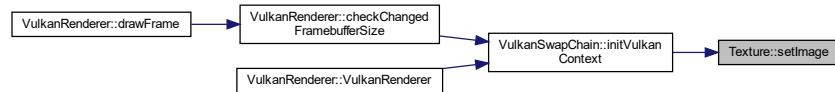
References [VulkanImage::setImage\(\)](#), and [vulkanImage](#).

Referenced by [VulkanSwapChain::initVulkanContext\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.34.3.13 setImageView()

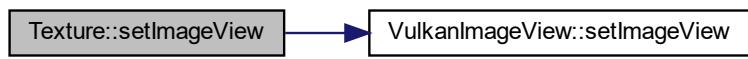
```
void Texture::setImageView (
    VkImageView imageView )
```

Definition at line 65 of file [Texture.cpp](#).

```
00065
00066     vulkanImageView.setImageView\(imageView\);
00067 }
```

References [VulkanImageView::setImageView\(\)](#), and [vulkanImageView](#).

Here is the call graph for this function:



## 5.34.4 Field Documentation

### 5.34.4.1 commandBufferManager

```
CommandBufferManager Texture::commandBufferManager [private]
```

Definition at line 52 of file [Texture.hpp](#).

Referenced by [generateMipMaps\(\)](#).

### 5.34.4.2 mip\_levels

```
uint32_t Texture::mip_levels = 0 [private]
```

Definition at line 42 of file [Texture.hpp](#).

Referenced by [createFromFile\(\)](#), [createImage\(\)](#), [createImageView\(\)](#), [generateMipMaps\(\)](#), and [getMipLevel\(\)](#).

### 5.34.4.3 vulkanBufferManager

```
VulkanBufferManager Texture::vulkanBufferManager [private]
```

Definition at line 53 of file [Texture.hpp](#).

Referenced by [createFromFile\(\)](#).

### 5.34.4.4 vulkanImage

```
VulkanImage Texture::vulkanImage [private]
```

Definition at line 55 of file [Texture.hpp](#).

Referenced by [cleanUp\(\)](#), [createFromFile\(\)](#), [createImage\(\)](#), [createImageView\(\)](#), [getImage\(\)](#), [getVulkanImage\(\)](#), and [setImage\(\)](#).

#### 5.34.4.5 vulkanImageView

VulkanImageView Texture::vulkanImageView [private]

Definition at line 56 of file [Texture.hpp](#).

Referenced by `cleanUp()`, `createImageView()`, `getImageView()`, `getVulkanImageView()`, and `setImageView()`.

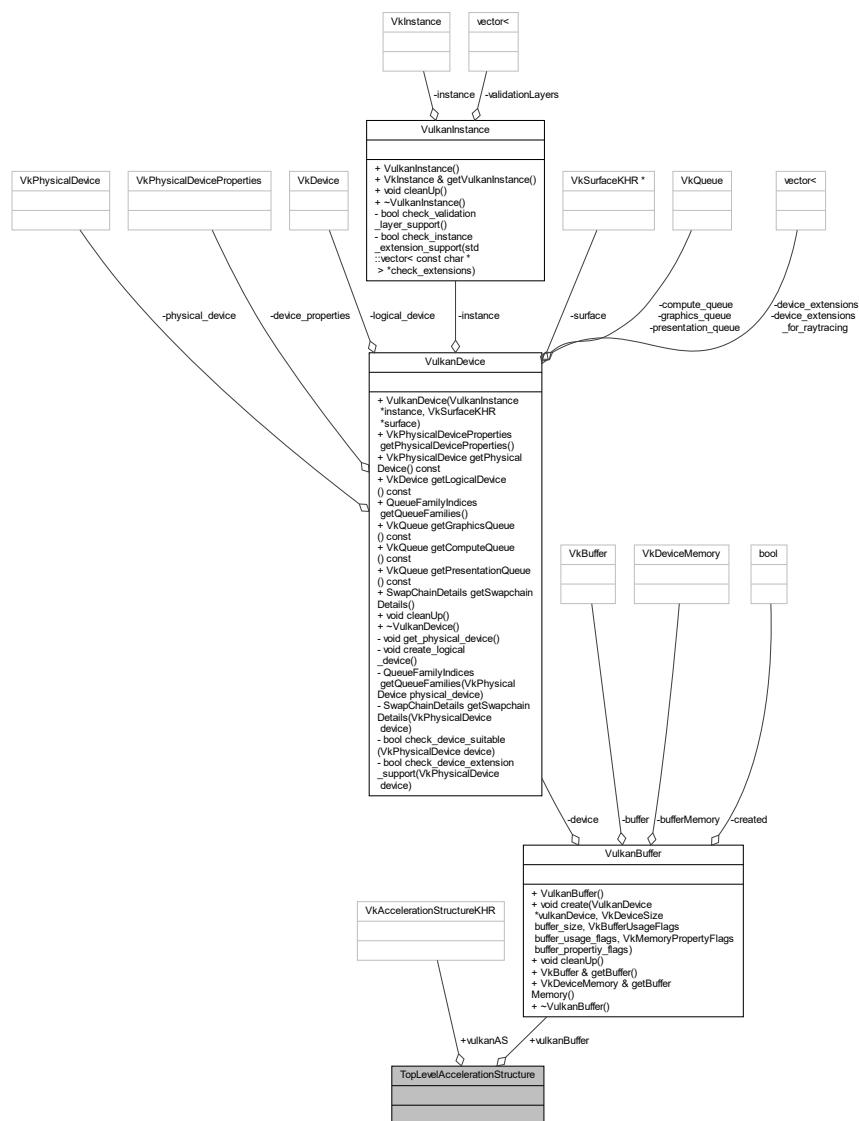
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/**Texture.hpp**
  - C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/**Texture.cpp**

## 5.35 TopLevelAccelerationStructure Struct Reference

```
#include <TopLevelAccelerationStructure.hpp>
```

## Collaboration diagram for TopLevelAccelerationStructure:



## Data Fields

- `VkAccelerationStructureKHR` [vulkanAS](#)
- `VulkanBuffer` [vulkanBuffer](#)

### 5.35.1 Detailed Description

Definition at line [7](#) of file [TopLevelAccelerationStructure.hpp](#).

### 5.35.2 Field Documentation

#### 5.35.2.1 `vulkanAS`

`VkAccelerationStructureKHR` [TopLevelAccelerationStructure::vulkanAS](#)

Definition at line [8](#) of file [TopLevelAccelerationStructure.hpp](#).

Referenced by [ASManager::cleanUp\(\)](#), [ASManager::createTLAS\(\)](#), and [ASManager::getTLAS\(\)](#).

#### 5.35.2.2 `vulkanBuffer`

`VulkanBuffer` [TopLevelAccelerationStructure::vulkanBuffer](#)

Definition at line [9](#) of file [TopLevelAccelerationStructure.hpp](#).

Referenced by [ASManager::cleanUp\(\)](#), and [ASManager::createTLAS\(\)](#).

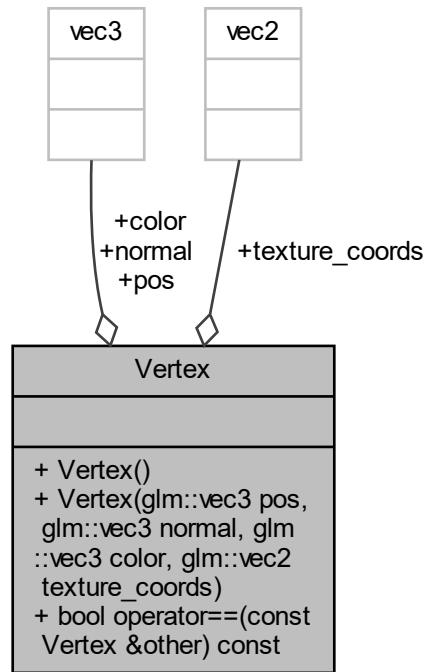
The documentation for this struct was generated from the following file:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/[TopLevelAccelerationStructure.hpp](#)

## 5.36 Vertex Struct Reference

```
#include <Vertex.hpp>
```

Collaboration diagram for Vertex:



### Public Member Functions

- [Vertex \(\)](#)
- [Vertex \(glm::vec3 pos, glm::vec3 normal, glm::vec3 color, glm::vec2 texture\\_coords\)](#)
- [bool operator==\(const Vertex &other\) const](#)

### Data Fields

- [glm::vec3 pos](#)
- [glm::vec3 normal](#)
- [glm::vec3 color](#)
- [glm::vec2 texture\\_coords](#)
- [vec3 pos](#)
- [vec3 normal](#)
- [vec3 color](#)
- [vec2 texture\\_coords](#)

### 5.36.1 Detailed Description

Definition at line 15 of file [Vertex.hpp](#).

### 5.36.2 Constructor & Destructor Documentation

#### 5.36.2.1 Vertex() [1/2]

```
Vertex::Vertex ( )
```

Definition at line 3 of file [Vertex.cpp](#).

```
00003     {
00004     this->pos = glm::vec3(-1.f);
00005     this->normal = glm::vec3(-1.f);
00006     this->color = glm::vec3(-1.f);
00007     this->texture_coords = glm::vec3(-1.f);
00008 }
```

References [color](#), [normal](#), [pos](#), and [texture\\_coords](#).

#### 5.36.2.2 Vertex() [2/2]

```
Vertex::Vertex (
    glm::vec3 pos,
    glm::vec3 normal,
    glm::vec3 color,
    glm::vec2 texture_coords )
```

Definition at line 10 of file [Vertex.cpp](#).

```
00011     {
00012     this->pos = pos;
00013     this->normal = normal;
00014     this->color = color;
00015     this->texture_coords = texture_coords;
00016 }
```

References [color](#), [normal](#), [pos](#), and [texture\\_coords](#).

### 5.36.3 Member Function Documentation

#### 5.36.3.1 operator==()

```
bool Vertex::operator== (
    const Vertex & other ) const [inline]
```

Definition at line 26 of file [Vertex.hpp](#).

```
00026     {
00027     return pos == other.pos && normal == other.normal &&
00028         texture_coords == other.texture_coords;
00029 }
```

References [normal](#), [pos](#), and [texture\\_coords](#).

## 5.36.4 Field Documentation

### 5.36.4.1 color [1/2]

```
glm::vec3 Vertex::color
```

Definition at line 23 of file [Vertex.hpp](#).

Referenced by [Vertex\(\)](#).

### 5.36.4.2 color [2/2]

```
vec3 Vertex::color
```

Definition at line 55 of file [Vertex.hpp](#).

### 5.36.4.3 normal [1/2]

```
glm::vec3 Vertex::normal
```

Definition at line 22 of file [Vertex.hpp](#).

Referenced by [ObjLoader::loadVertices\(\)](#), [operator==\(\)](#), and [Vertex\(\)](#).

### 5.36.4.4 normal [2/2]

```
vec3 Vertex::normal
```

Definition at line 54 of file [Vertex.hpp](#).

### 5.36.4.5 pos [1/2]

```
glm::vec3 Vertex::pos
```

Definition at line 21 of file [Vertex.hpp](#).

Referenced by [ObjLoader::loadVertices\(\)](#), [operator==\(\)](#), and [Vertex\(\)](#).

#### 5.36.4.6 pos [2/2]

```
vec3 Vertex::pos
```

Definition at line [53](#) of file [Vertex.hpp](#).

#### 5.36.4.7 texture\_coords [1/2]

```
glm::vec2 Vertex::texture_coords
```

Definition at line [24](#) of file [Vertex.hpp](#).

Referenced by [operator==\(\)](#), and [Vertex\(\)](#).

#### 5.36.4.8 texture\_coords [2/2]

```
vec2 Vertex::texture_coords
```

Definition at line [56](#) of file [Vertex.hpp](#).

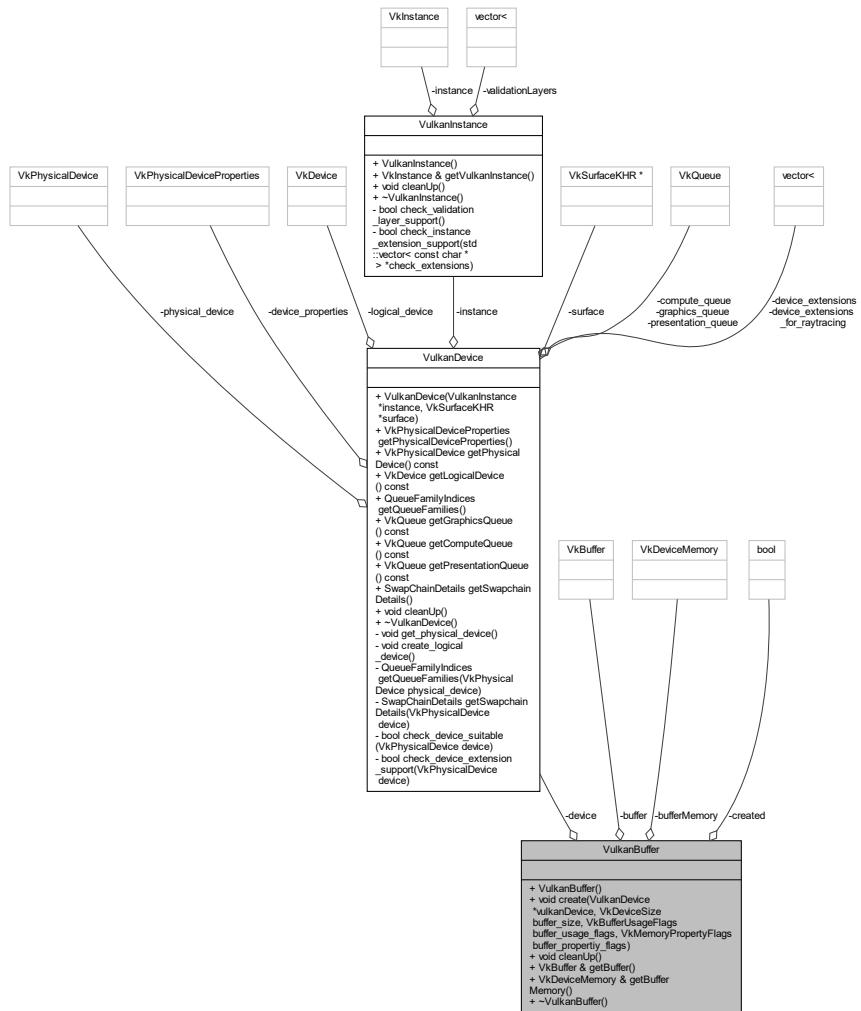
The documentation for this struct was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/[Vertex.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/[Vertex.cpp](#)

## 5.37 VulkanBuffer Class Reference

```
#include <VulkanBuffer.hpp>
```

Collaboration diagram for VulkanBuffer:



## Public Member Functions

- [VulkanBuffer \(\)](#)
- void [create \(VulkanDevice \\*vulkanDevice, VkDeviceSize buffer\\_size, VkBufferUsageFlags buffer\\_usage\\_flags, VkMemoryPropertyFlags buffer\\_property\\_flags\)](#)
- void [cleanUp \(\)](#)
- [VkBuffer & getBuffer \(\)](#)
- [VkDeviceMemory & getBufferMemory \(\)](#)
- [~VulkanBuffer \(\)](#)

## Private Attributes

- `VulkanDevice * device {VK_NULL_HANDLE}`
- `VkBuffer buffer {VK_NULL_HANDLE}`
- `VkDeviceMemory bufferMemory {VK_NULL_HANDLE}`
- `bool created {false}`

### 5.37.1 Detailed Description

Definition at line 6 of file [VulkanBuffer.hpp](#).

### 5.37.2 Constructor & Destructor Documentation

#### 5.37.2.1 VulkanBuffer()

```
VulkanBuffer::VulkanBuffer ( )
```

Definition at line 8 of file [VulkanBuffer.cpp](#).  
00008 { }

#### 5.37.2.2 ~VulkanBuffer()

```
VulkanBuffer::~VulkanBuffer ( )
```

Definition at line 70 of file [VulkanBuffer.cpp](#).  
00070 { }

### 5.37.3 Member Function Documentation

#### 5.37.3.1 cleanUp()

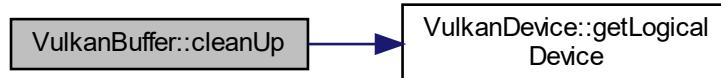
```
void VulkanBuffer::cleanUp ( )
```

Definition at line 63 of file [VulkanBuffer.cpp](#).  
00063 {  
00064 if ([created](#)) {  
00065 [vkDestroyBuffer](#)([device](#)->[getLogicalDevice](#)(), [buffer](#), [nullptr](#));  
00066 [vkFreeMemory](#)([device](#)->[getLogicalDevice](#)(), [bufferMemory](#), [nullptr](#));  
00067 }  
00068 }

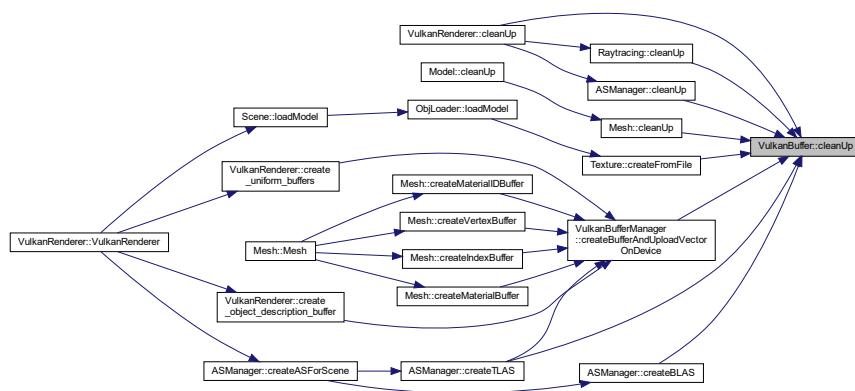
References [buffer](#), [bufferMemory](#), [created](#), [device](#), and [VulkanDevice::getLogicalDevice\(\)](#).

Referenced by [ASManager::cleanUp\(\)](#), [Raytracing::cleanUp\(\)](#), [VulkanRenderer::cleanUp\(\)](#), [Mesh::cleanUp\(\)](#), [ASManager::createBLAS\(\)](#), [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [Texture::createFromFile\(\)](#), and [ASManager::createTLAS\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.37.3.2 create()

```

void VulkanBuffer::create (
    VulkanDevice * vulkanDevice,
    VkDeviceSize buffer_size,
    VkBufferUsageFlags buffer_usage_flags,
    VkMemoryPropertyFlags buffer_property_flags )
  
```

Definition at line 10 of file [VulkanBuffer.cpp](#).

```

00012
00013     this->device = device;
00014
00015     // information to create a buffer (doesn't include assigning memory)
00016     VkBufferCreateInfo buffer_info{};
00017     buffer_info.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
00018     buffer_info.size = buffer_size;
00019     // multiple types of buffer possible, e.g. vertex buffer
00020     buffer_info.usage = buffer_usage_flags;
00021     // similar to swap chain images, can share vertex buffers
00022     buffer_info.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
00023
00024     VkResult result = vkCreateBuffer(device->getLogicalDevice(), &buffer_info,
00025                                         nullptr, &buffer);
00026     ASSERT_VULKAN(result, "Failed to create a buffer!");
00027
00028     // get buffer memory requirements
00029     VkMemoryRequirements memory_requirements{};
  
```

```

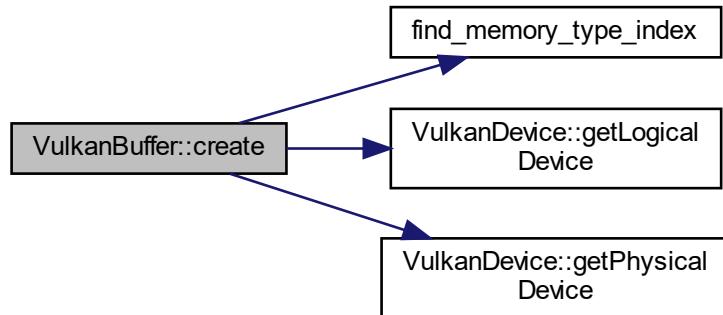
00030    vkGetBufferMemoryRequirements(device->getLogicalDevice(), buffer,
00031                                &memory_requirements);
00032
00033    // allocate memory to buffer
00034    VkMemoryAllocateInfo memory_alloc_info{};
00035    memory_alloc_info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
00036    memory_alloc_info.allocationSize = memory_requirements.size;
00037
00038    uint32_t memory_type_index = find_memory_type_index(
00039        device->getPhysicalDevice(), memory_requirements.memoryTypeBits,
00040        buffer_property_flags);
00041
00042    // VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | /* memory is visible to
00043    // CPU side
00044    // */ VK_MEMORY_PROPERTY_HOST_COHERENT_BIT /* data is placed straight into
00045    // buffer */;
00046    if (memory_type_index < 0) {
00047        throw std::runtime_error("Failed to find suitable memory type!");
00048    }
00049
00050    memory_alloc_info.memoryTypeIndex = memory_type_index;
00051
00052    // allocate memory to VkDeviceMemory
00053    result = vkAllocateMemory(device->getLogicalDevice(), &memory_alloc_info,
00054                             nullptr, &bufferMemory);
00055    ASSERT_VULKAN(result, "Failed to allocate memory for buffer!");
00056
00057    // allocate memory to given buffer
00058    vkBindBufferMemory(device->getLogicalDevice(), buffer, bufferMemory, 0);
00059
00060    created = true;
00061 }

```

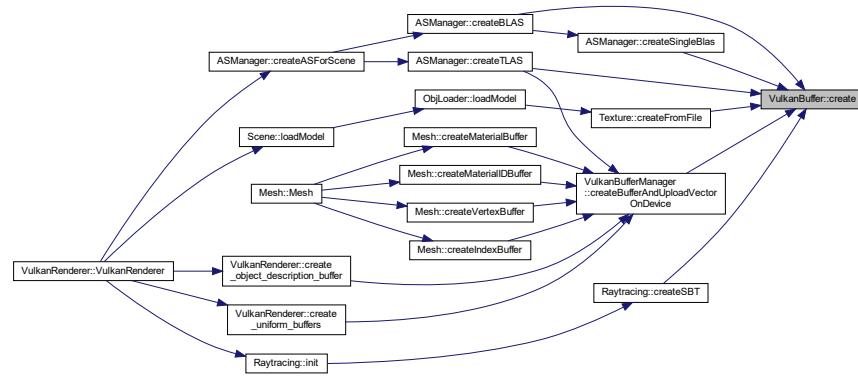
References [buffer](#), [bufferMemory](#), [created](#), [device](#), [find\\_memory\\_type\\_index\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), and [VulkanDevice::getPhysicalDevice\(\)](#).

Referenced by [ASManager::createBLAS\(\)](#), [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [Texture::createFromFile\(\)](#), [Raytracing::createSBT\(\)](#), [ASManager::createSingleBlas\(\)](#), and [ASManager::createTLAS\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.37.3.3 getBuffer()

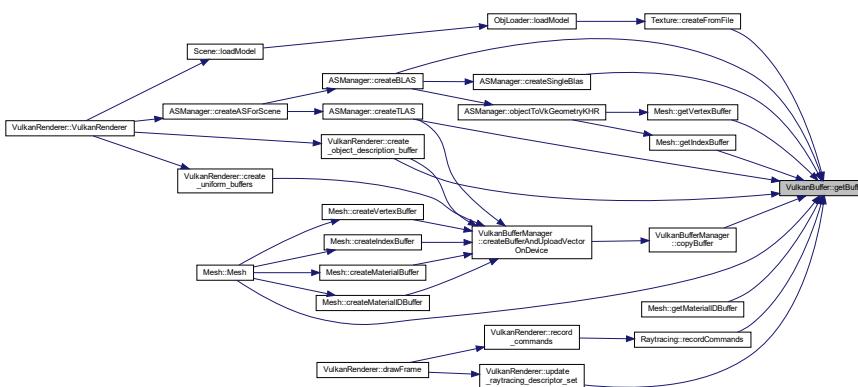
```
VkBuffer & VulkanBuffer::getBuffer( ) [inline]
```

Definition at line 16 of file [VulkanBuffer.hpp](#).  
00016 { **return** buffer; ; }

References [buffer](#).

Referenced by [VulkanBufferManager::copyBuffer\(\)](#), [VulkanRenderer::create\\_object\\_description\\_buffer\(\)](#), [ASManager::createBLAS\(\)](#), [Texture::createFromFile\(\)](#), [ASManager::createSingleBlas\(\)](#), [ASManager::createTLAS\(\)](#), [Mesh::getIndexBuffer\(\)](#), [Mesh::getMaterialIDBuffer\(\)](#), [Mesh::getVertexBuffer\(\)](#), [Mesh::Mesh\(\)](#), [Raytracing::recordCommands\(\)](#), and [VulkanRenderer::update\\_raytracing\\_descriptor\\_set\(\)](#).

Here is the caller graph for this function:



### 5.37.3.4 `getBufferMemory()`

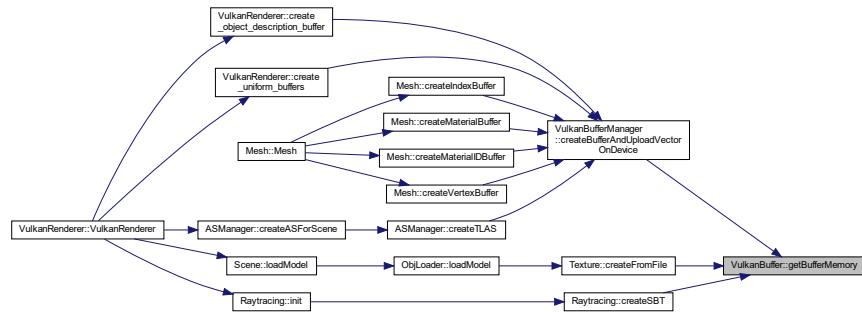
```
VkDeviceMemory & VulkanBuffer::getBufferMemory () [inline]
```

Definition at line 17 of file [VulkanBuffer.hpp](#).  
 00017 { **return** bufferMemory; };

References [bufferMemory](#).

Referenced by [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [Texture::createFromFile\(\)](#), and [Raytracing::createSBT\(\)](#).

Here is the caller graph for this function:



## 5.37.4 Field Documentation

### 5.37.4.1 `buffer`

```
VkBuffer VulkanBuffer::buffer {VK_NULL_HANDLE} [private]
```

Definition at line 24 of file [VulkanBuffer.hpp](#).

Referenced by [cleanUp\(\)](#), [create\(\)](#), and [getBuffer\(\)](#).

### 5.37.4.2 `bufferMemory`

```
VkDeviceMemory VulkanBuffer::bufferMemory {VK_NULL_HANDLE} [private]
```

Definition at line 25 of file [VulkanBuffer.hpp](#).

Referenced by [cleanUp\(\)](#), [create\(\)](#), and [getBufferMemory\(\)](#).

### 5.37.4.3 created

```
bool VulkanBuffer::created {false} [private]
```

Definition at line 27 of file [VulkanBuffer.hpp](#).

Referenced by [cleanUp\(\)](#), and [create\(\)](#).

### 5.37.4.4 device

```
VulkanDevice* VulkanBuffer::device {VK_NULL_HANDLE} [private]
```

Definition at line 22 of file [VulkanBuffer.hpp](#).

Referenced by [cleanUp\(\)](#), and [create\(\)](#).

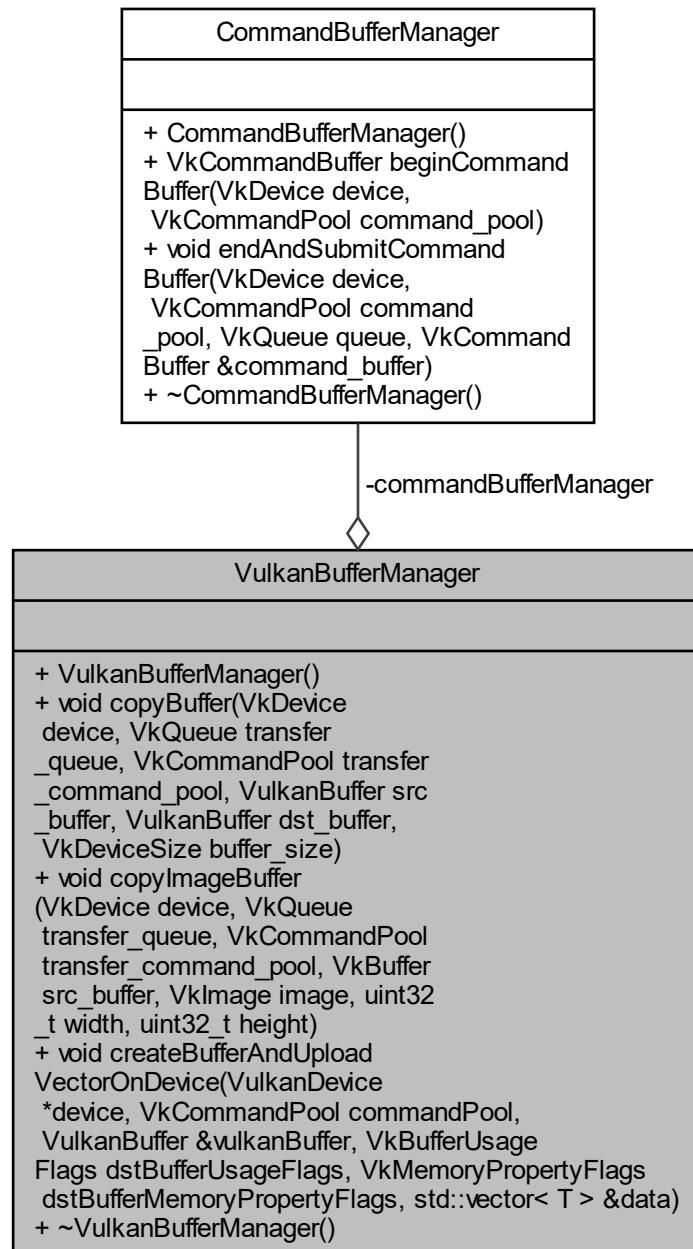
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/[VulkanBuffer.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/[VulkanBuffer.cpp](#)

## 5.38 VulkanBufferManager Class Reference

```
#include <VulkanBufferManager.hpp>
```

Collaboration diagram for VulkanBufferManager:



## Public Member Functions

- [`VulkanBufferManager \(\)`](#)
- void [`copyBuffer`](#) (`VkDevice device, VkQueue transfer_queue, VkCommandPool transfer_command_pool, VulkanBuffer src_buffer, VulkanBuffer dst_buffer, VkDeviceSize buffer_size`)
- void [`copyImageBuffer`](#) (`VkDevice device, VkQueue transfer_queue, VkCommandPool transfer_command_pool, VkBuffer src_buffer, VklImage image, uint32_t width, uint32_t height`)

- template<typename T >  
void `createBufferAndUploadVectorOnDevice` (`VulkanDevice` \*device, `VkCommandPool` commandPool,  
`VulkanBuffer` &vulkanBuffer, `VkBufferUsageFlags` dstBufferUsageFlags, `VkMemoryPropertyFlags` dstBufferMemoryPropertyFlags, `std::vector< T >` &data)
- `~VulkanBufferManager` ()

## Private Attributes

- `CommandBufferManager` commandBufferManager

### 5.38.1 Detailed Description

Definition at line 11 of file [VulkanBufferManager.hpp](#).

### 5.38.2 Constructor & Destructor Documentation

#### 5.38.2.1 `VulkanBufferManager()`

`VulkanBufferManager::VulkanBufferManager` ( )

Definition at line 3 of file [VulkanBufferManager.cpp](#).  
00003 { }

#### 5.38.2.2 `~VulkanBufferManager()`

`VulkanBufferManager::~VulkanBufferManager` ( )

Definition at line 59 of file [VulkanBufferManager.cpp](#).  
00059 { }

### 5.38.3 Member Function Documentation

### 5.38.3.1 copyBuffer()

```
void VulkanBufferManager::copyBuffer (
    VkDevice device,
    VkQueue transfer_queue,
    VkCommandPool transfer_command_pool,
    VulkanBuffer src_buffer,
    VulkanBuffer dst_buffer,
    VkDeviceSize buffer_size )
```

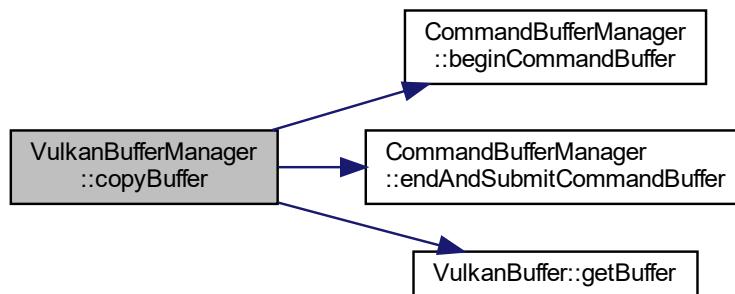
Definition at line 5 of file [VulkanBufferManager.cpp](#).

```
00009
00010     // create buffer
00011     VkCommandBuffer command_buffer =
00012         commandBufferManager.beginCommandBuffer(device, transfer_command_pool);
00013
00014     // region of data to copy from and to
00015     VkBufferCopy buffer_copy_region{};
00016     buffer_copy_region.srcOffset = 0;
00017     buffer_copy_region.dstOffset = 0;
00018     buffer_copy_region.size = buffer_size;
00019
00020     // command to copy src buffer to dst buffer
00021     vkCmdCopyBuffer(command_buffer, src_buffer.getBuffer(),
00022                      dst_buffer.getBuffer(), 1, &buffer_copy_region);
00023
00024     commandBufferManager.endAndSubmitCommandBuffer(
00025         device, transfer_command_pool, transfer_queue, command_buffer);
00026 }
```

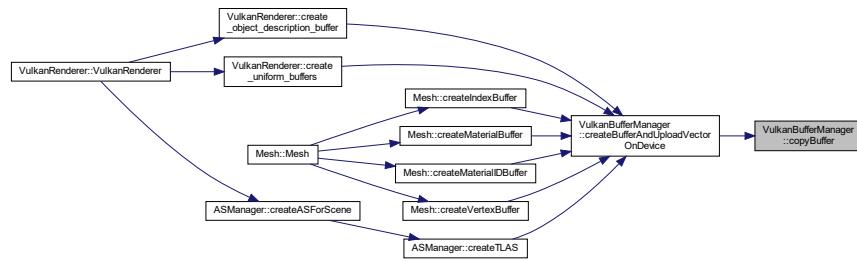
References [CommandBufferManager::beginCommandBuffer\(\)](#), [commandBufferManager](#), [CommandBufferManager::endAndSubmitC](#) and [VulkanBuffer::getBuffer\(\)](#).

Referenced by [createCommandBufferAndUploadVectorOnDevice\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.38.3.2 copyImageBuffer()

```
void VulkanBufferManager::copyImageBuffer (
    VkDevice device,
    VkQueue transfer_queue,
    VkCommandPool transfer_command_pool,
    VkBuffer src_buffer,
    VkImage image,
    uint32_t width,
    uint32_t height )
```

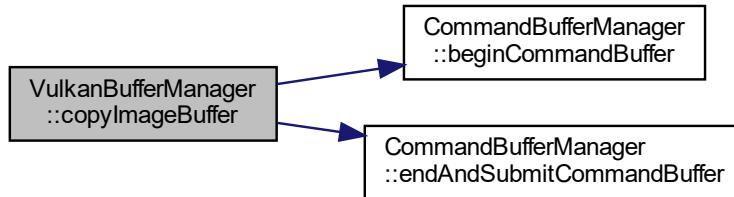
Definition at line 28 of file [VulkanBufferManager.cpp](#).

```
00032
00033 // create buffer
00034 VkCommandBuffer transfer_command_buffer =
00035     commandBufferManager.beginCommandBuffer(device, transfer_command_pool);
00036
00037 VkBufferImageCopy image_region{};
00038 image_region.bufferOffset = 0; // offset into data
00039 image_region.bufferRowLength =
00040     0; // row length of data to calculate data spacing
00041 image_region.bufferImageHeight = 0; // image height to calculate data spacing
00042 image_region.imageSubresource.aspectMask =
00043     VK_IMAGE_ASPECT_COLOR_BIT; // which aspect of image to copy
00044 image_region.imageSubresource.mipLevel = 0;
00045 image_region.imageSubresource.baseArrayLayer = 0;
00046 image_region.imageSubresource.layerCount = 1;
00047 image_region.imageOffset = {0, 0, 0}; // offset into image
00048 image_region.imageExtent = {width, height, 1};
00049
00050 // copy buffer to given image
00051 vkCmdCopyBufferToImage(transfer_command_buffer, src_buffer, image,
00052                         VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1,
00053                         &image_region);
00054
00055 commandBufferManager.endAndSubmitCommandBuffer(
00056     device, transfer_command_pool, transfer_queue, transfer_command_buffer);
00057 }
```

References [CommandBufferManager::beginCommandBuffer\(\)](#), [commandBufferManager](#), and [CommandBufferManager::endAndSubmitCommandBuffer\(\)](#).

Referenced by [Texture::createFromFile\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.38.3.3 createBufferAndUploadVectorOnDevice()

```

template<typename T >
void VulkanBufferManager::createBufferAndUploadVectorOnDevice (
    VulkanDevice * device,
    VkCommandPool commandPool,
    VulkanBuffer & vulkanBuffer,
    VkBufferUsageFlags dstBufferUsageFlags,
    VkMemoryPropertyFlags dstBufferMemoryPropertyFlags,
    std::vector< T > & data ) [inline]

```

Definition at line 36 of file [VulkanBufferManager.hpp](#).

```

00040     {
00041     VkDeviceSize bufferSize = sizeof(T) * bufferData.size();
00042
00043     // temporary buffer to "stage" vertex data before transferring to GPU
00044     VulkanBuffer stagingBuffer;
00045
00046     // create buffer and allocate memory to it
00047     stagingBuffer.create(device, bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
00048                           VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
00049                           VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);
00050
00051     // Map memory to vertex buffer
00052     // 1.) create pointer to a point in normal memory
00053     void* data;
00054     // 2.) map the vertex buffer memory to that point
00055     vkMapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory(), 0,
00056                 bufferSize, 0, &data);
00057     // 3.) copy memory from vertices vector to the point
00058     std::memcpy(data, bufferData.data(), static_cast<size_t>(bufferSize));
00059     // 4.) unmap the vertex buffer memory
00060     vkUnmapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory());
00061
00062     // create buffer with TRANSFER_DST_BIT to mark as recipient of transfer data
00063     // (also VERTEX_BUFFER) buffer memory is to be DEVICE_LOCAL_BIT meaning memory
00064     // is on the GPU and only accessible by it and not CPU (host)

```

```

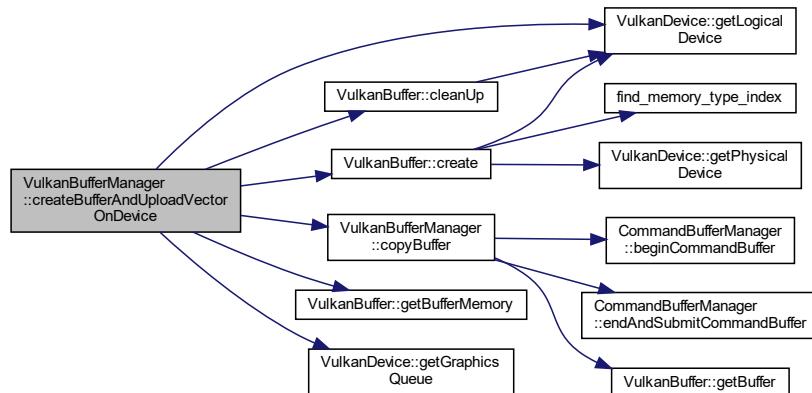
00065     vulkanBuffer.create(device, bufferSize, dstBufferUsageFlags,
00066                             dstBufferMemoryPropertyFlags);
00067
00068     // copy staging buffer to vertex buffer on GPU
00069     copyBuffer(device->getLogicalDevice(), device->getGraphicsQueue(),
00070                commandPool, stagingBuffer, vulkanBuffer, bufferSize);
00071
00072     stagingBuffer.cleanUp();
00073 }

```

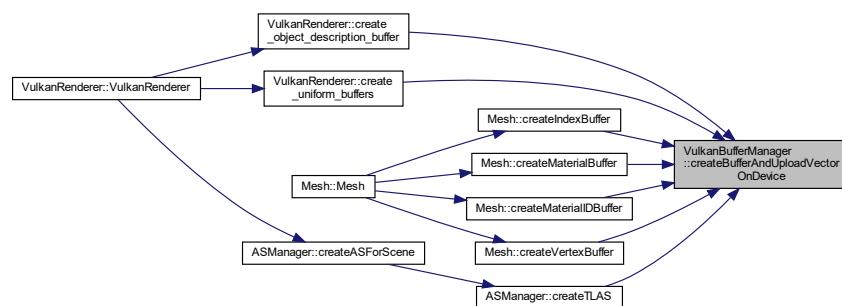
References [VulkanBuffer::cleanUp\(\)](#), [copyBuffer\(\)](#), [VulkanBuffer::create\(\)](#), [VulkanBuffer::getBufferMemory\(\)](#), [VulkanDevice::getGraphicsQueue\(\)](#), and [VulkanDevice::getLogicalDevice\(\)](#).

Referenced by [VulkanRenderer::create\\_object\\_description\\_buffer\(\)](#), [VulkanRenderer::create\\_uniform\\_buffers\(\)](#), [Mesh::createIndexBuffer\(\)](#), [Mesh::createMaterialBuffer\(\)](#), [Mesh::createMaterialIDBuffer\(\)](#), [ASManager::createTLAS\(\)](#), and [Mesh::createVertexBuffer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.38.4 Field Documentation

### 5.38.4.1 commandBufferManager

```
CommandBufferManager VulkanBufferManager::commandBufferManager [private]
```

Definition at line 32 of file [VulkanBufferManager.hpp](#).

Referenced by [copyBuffer\(\)](#), and [copyImageBuffer\(\)](#).

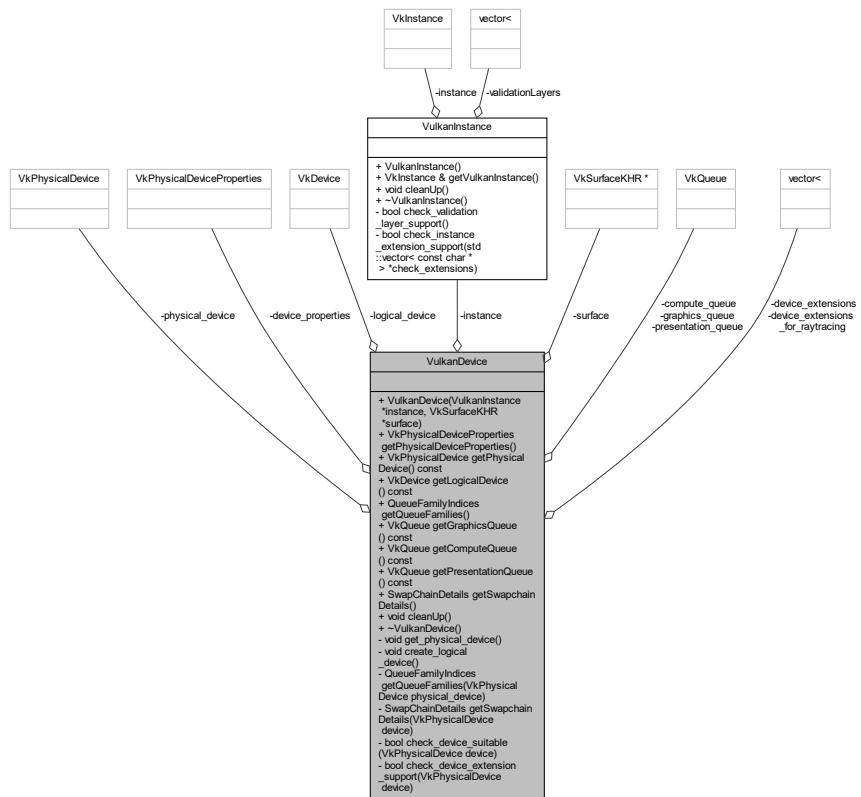
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/[VulkanBufferManager.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/[VulkanBufferManager.cpp](#)

## 5.39 VulkanDevice Class Reference

```
#include <VulkanDevice.hpp>
```

Collaboration diagram for VulkanDevice:



## Public Member Functions

- `VulkanDevice (VulkanInstance *instance, VkSurfaceKHR *surface)`
- `VkPhysicalDeviceProperties getPhysicalDeviceProperties ()`
- `VkPhysicalDevice getPhysicalDevice () const`
- `VkDevice getLogicalDevice () const`
- `QueueFamilyIndices getQueueFamilies ()`
- `VkQueue getGraphicsQueue () const`
- `VkQueue getComputeQueue () const`
- `VkQueue getPresentationQueue () const`
- `SwapChainDetails getSwapchainDetails ()`
- `void cleanUp ()`
- `~VulkanDevice ()`

## Private Member Functions

- `void get_physical_device ()`
- `void create_logical_device ()`
- `QueueFamilyIndices getQueueFamilies (VkPhysicalDevice physical_device)`
- `SwapChainDetails getSwapchainDetails (VkPhysicalDevice device)`
- `bool check_device_suitable (VkPhysicalDevice device)`
- `bool check_device_extension_support (VkPhysicalDevice device)`

## Private Attributes

- `VkPhysicalDevice physical_device`
- `VkPhysicalDeviceProperties device_properties`
- `VkDevice logical_device`
- `VulkanInstance * instance`
- `VkSurfaceKHR * surface`
- `VkQueue graphics_queue`
- `VkQueue presentation_queue`
- `VkQueue compute_queue`
- `const std::vector< const char * > device_extensions`
- `const std::vector< const char * > device_extensions_for_raytracing`

### 5.39.1 Detailed Description

Definition at line 10 of file [VulkanDevice.hpp](#).

### 5.39.2 Constructor & Destructor Documentation

### 5.39.2.1 VulkanDevice()

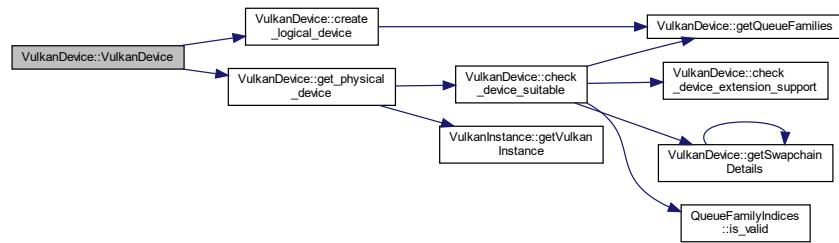
```
VulkanDevice::VulkanDevice (
    VulkanInstance * instance,
    VkSurfaceKHR * surface )
```

Definition at line 8 of file [VulkanDevice.cpp](#).

```
00008     this->instance = instance;
00009     this->surface = surface;
00010     get_physical_device();
00011     create_logical_device();
00012 }
00013 }
```

References [create\\_logical\\_device\(\)](#), [get\\_physical\\_device\(\)](#), [instance](#), and [surface](#).

Here is the call graph for this function:



### 5.39.2.2 ~VulkanDevice()

```
VulkanDevice::~VulkanDevice ( )
```

Definition at line 21 of file [VulkanDevice.cpp](#).

```
00021 { }
```

## 5.39.3 Member Function Documentation

### 5.39.3.1 check\_device\_extension\_support()

```
bool VulkanDevice::check_device_extension_support (
    VkPhysicalDevice device ) [private]
```

Definition at line 346 of file [VulkanDevice.cpp](#).

```
00346
00347     uint32_t extension_count = 0;
00348     vkEnumerateDeviceExtensionProperties(device, nullptr, &extension_count,
00349                                             nullptr);
00350
00351     if (extension_count == 0) {
00352         return false;
00353     }
00354
00355     // populate list of extensions
00356     std::vector<VkExtensionProperties> extensions(extension_count);
00357     vkEnumerateDeviceExtensionProperties(device, nullptr, &extension_count,
00358                                         extensions.data());
00359
00360     for (const auto& device_extension : device_extensions) {
00361         bool has_extension = false;
00362
00363         for (const auto& extension : extensions) {
00364             if (strcmp(device_extension, extension.extensionName) == 0) {
00365                 has_extension = true;
00366                 break;
00367             }
00368         }
00369
00370         if (!has_extension) {
00371             return false;
00372         }
00373     }
00374
00375     return true;
00376 }
```

References [device\\_extensions](#).

Referenced by [check\\_device\\_suitable\(\)](#).

Here is the caller graph for this function:



### 5.39.3.2 check\_device\_suitable()

```
bool VulkanDevice::check_device_suitable (
    VkPhysicalDevice device ) [private]
```

Definition at line 322 of file [VulkanDevice.cpp](#).

```
00322
00323     // Information about device itself (ID, name, type, vendor, etc)
00324     VkPhysicalDeviceProperties device_properties;
00325     vkGetPhysicalDeviceProperties(device, &device_properties);
00326
00327     VkPhysicalDeviceFeatures device_features;
00328     vkGetPhysicalDeviceFeatures(device, &device_features);
00329
00330     QueueFamilyIndices indices = getQueueFamilies(device);
00331
00332     bool extensions_supported = check_device_extension_support(device);
```

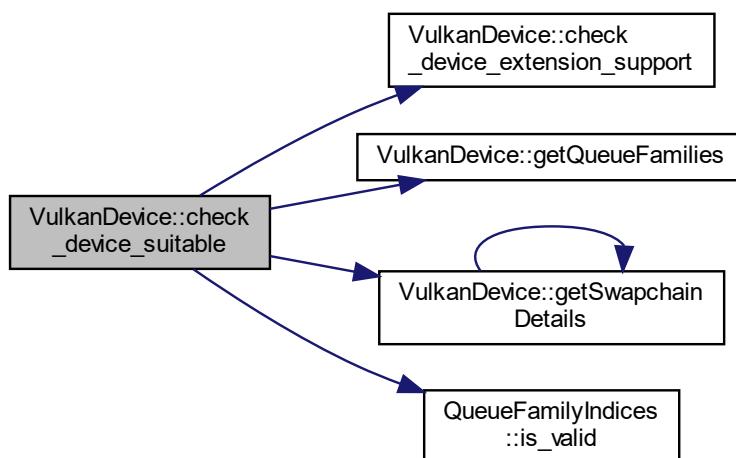
```

00333     bool swap_chain_valid = false;
00334
00335     if (extensions_supported) {
00336         SwapChainDetails swap_chain_details = getSwapchainDetails(device);
00337         swap_chain_valid = !swap_chain_details.presentation_mode.empty() &&
00338                         !swap_chain_details.formats.empty();
00339     }
00340 }
00341
00342     return indices.is_valid() && extensions_supported && swap_chain_valid &&
00343         device_features.samplerAnisotropy;
00344 }
```

References [check\\_device\\_extension\\_support\(\)](#), [device\\_properties](#), [SwapChainDetails::formats](#), [getQueueFamilies\(\)](#), [getSwapchainDetails\(\)](#), [QueueFamilyIndices::is\\_valid\(\)](#), and [SwapChainDetails::presentation\\_mode](#).

Referenced by [get\\_physical\\_device\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.39.3.3 cleanUp()

```
void VulkanDevice::cleanUp ( )
```

Definition at line 19 of file [VulkanDevice.cpp](#).

```
00019 { vkDestroyDevice(logical_device, nullptr); }
```

References [logical\\_device](#).

### 5.39.3.4 create\_logical\_device()

```
void VulkanDevice::create_logical_device ( ) [private]
```

Definition at line 100 of file [VulkanDevice.cpp](#).

```
00100     {
00101     // get the queue family indices for the chosen physical device
00102     QueueFamilyIndices indices = getQueueFamilies();
00103
00104     // vector for queue creation information and set for family indices
00105     std::vector<VkDeviceQueueCreateInfo> queue_create_infos;
00106     std::set<int> queue_family_indices = {indices.graphics_family,
00107                                         indices.presentation_family,
00108                                         indices.compute_family};
00109
00110     // Queue the logical device needs to create info to do so (only 1 for now,
00111     // will add more later!)
00112     for (int queue_family_index : queue_family_indices) {
00113         VkDeviceQueueCreateInfo queue_create_info{};
00114         queue_create_info.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
00115         queue_create_info.queueFamilyIndex =
00116             queue_family_index; // the index of the family to create a queue from
00117         queue_create_info.queueCount = 1; // number of queues to create
00118         float priority = 1.0f;
00119         queue_create_info.pQueuePriorities =
00120             &priority; // Vulkan needs to know how to handle multiple queues, so
00121             // decide priority (1 = highest)
00122
00123         queue_create_infos.push_back(queue_create_info);
00124     }
00125
00126     // -- ALL EXTENSION WE NEED
00127     VkPhysicalDeviceDescriptorIndexingFeatures indexing_features{};
00128     indexing_features.sType =
00129         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES;
00130     indexing_features.runtimeDescriptorArray = VK_TRUE;
00131     indexing_features.shaderSampledImageArrayNonUniformIndexing = VK_TRUE;
00132     indexing_features.pNext = nullptr;
00133
00134     // -- NEEDED FOR QUERING THE DEVICE ADDRESS WHEN CREATING ACCELERATION
00135     // STRUCTURES
00136     VkPhysicalDeviceBufferDeviceAddressFeaturesEXT
00137         buffer_device_address_features{};
00138     buffer_device_address_features.sType =
00139         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT;
00140     buffer_device_address_features.pNext = &indexing_features;
00141     buffer_device_address_features.bufferDeviceAddress = VK_TRUE;
00142     buffer_device_address_features.bufferDeviceAddressCaptureReplay = VK_TRUE;
00143     buffer_device_address_features.bufferDeviceAddressMultiDevice = VK_FALSE;
00144
00145     // --ENABLE RAY TRACING PIPELINE
00146     VkPhysicalDeviceRayTracingPipelineFeaturesKHR ray_tracing_pipeline_features{};
00147     ray_tracing_pipeline_features.sType =
00148         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_FEATURES_KHR;
00149     ray_tracing_pipeline_features.pNext = &buffer_device_address_features;
00150     ray_tracing_pipeline_features.rayTracingPipeline = VK_TRUE;
00151
00152     // -- ENABLE ACCELERATION STRUCTURES
00153     VkPhysicalDeviceAccelerationStructureFeaturesKHR
00154         acceleration_structure_features{};
00155     acceleration_structure_features.sType =
00156         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_FEATURES_KHR;
00157     acceleration_structure_features.pNext = &ray_tracing_pipeline_features;
00158     acceleration_structure_features.accelerationStructure = VK_TRUE;
00159     acceleration_structure_features.accelerationStructureCaptureReplay = VK_TRUE;
00160     acceleration_structure_features.accelerationStructureIndirectBuild = VK_FALSE;
00161     acceleration_structure_features.accelerationStructureHostCommands = VK_FALSE;
00162     acceleration_structure_features
00163         .descriptorBindingAccelerationStructureUpdateAfterBind = VK_FALSE;
00164
00165     VkPhysicalDeviceVulkan13Features features13{};
00166     features13.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES;
00167     features13.maintenance4 = VK_TRUE;
00168     features13.robustImageAccess = VK_FALSE;
00169     features13.inlineUniformBlock = VK_FALSE;
00170     features13.descriptorBindingInlineUniformBlockUpdateAfterBind = VK_FALSE;
00171     features13.pipelineCreationCacheControl = VK_FALSE;
00172     features13.privateData = VK_FALSE;
00173     features13.shaderDemoteToHelperInvocation = VK_FALSE;
00174     features13.shaderTerminateInvocation = VK_FALSE;
00175     features13.subgroupSizeControl = VK_FALSE;
00176     features13.computeFullSubgroups = VK_FALSE;
00177     features13.synchronization2 = VK_FALSE;
00178     features13.textureCompressionASTC_HDR = VK_FALSE;
00179     features13.shaderZeroInitializeWorkgroupMemory = VK_FALSE;
```

```

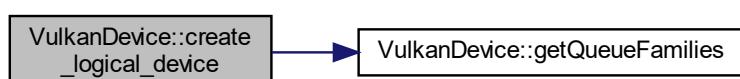
00180     features13.dynamicRendering = VK_FALSE;
00181     features13.shaderIntegerDotProduct = VK_FALSE;
00182     features13.pNext = &acceleration_structure_features;
00183
00184     VkPhysicalDeviceRayQueryFeaturesKHR rayQueryFeature{};
00185     rayQueryFeature.sType =
00186         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_QUERY_FEATURES_KHR;
00187     rayQueryFeature.pNext = &features13;
00188     rayQueryFeature.rayQuery = VK_TRUE;
00189
00190     VkPhysicalDeviceFeatures2 features2{};
00191     features2.pNext = &rayQueryFeature;
00192     features2.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2;
00193     features2.features.samplerAnisotropy = VK_TRUE;
00194     features2.features.shaderInt64 = VK_TRUE;
00195     features2.features.geometryShader = VK_TRUE;
00196     features2.features.logicOp = VK_TRUE;
00197
00198 // -- PREPARE FOR HAVING MORE EXTENSION BECAUSE WE NEED RAYTRACING
00199 // CAPABILITIES
00200 std::vector<const char*> extensions(device_extensions);
00201
00202 // COPY ALL NECESSARY EXTENSIONS FOR RAYTRACING TO THE EXTENSION
00203 extensions.insert(extensions.begin(),
00204     device_extensions_for_raytracing.begin(),
00205     device_extensions_for_raytracing.end());
00206
00207 // information to create logical device (sometimes called "device")
00208 VkDeviceCreateInfo device_create_info{};
00209 device_create_info.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
00210 device_create_info.queueCreateInfoCount = static_cast<uint32_t>(
00211     queue_create_infos.size()); // number of queue create infos
00212 device_create_info.pQueueCreateInfos =
00213     queue_create_infos.data(); // list of queue create infos so device can
00214     // create required queues
00215 device_create_info.enabledExtensionCount = static_cast<uint32_t>(
00216     extensions.size()); // number of enabled logical device extensions
00217 device_create_info.ppEnabledExtensionNames =
00218     extensions.data(); // list of enabled logical device extensions
00219 device_create_info.flags = 0;
00220 device_create_info.pEnabledFeatures = NULL;
00221
00222 device_create_info.pNext = &features2;
00223
00224 // create logical device for the given physical device
00225 VkResult result = vkCreateDevice(physical_device, &device_create_info,
00226                                     nullptr, &logical_device);
00227 ASSERT_VULKAN(result, "Failed to create a logical device!");
00228
00229 // Queues are created at the same time as the device...
00230 // So we want handle to queues
00231 // From given logical device of given queue family, of given queue index (0
00232 // since only one queue), place reference in given VkQueue
00233 vkGetDeviceQueue(logical_device, indices.graphics_family, 0, &graphics_queue);
00234 vkGetDeviceQueue(logical_device, indices.presentation_family, 0,
00235     &presentation_queue);
00236 vkGetDeviceQueue(logical_device, indices.compute_family, 0, &compute_queue);
00237 }

```

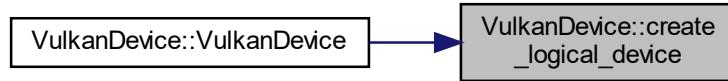
References QueueFamilyIndices::compute\_family, compute\_queue, device\_extensions, device\_extensions\_for\_raytracing, getQueueFamilies(), QueueFamilyIndices::graphics\_family, graphics\_queue, logical\_device, physical\_device, QueueFamilyIndices::presentation\_family, and presentation\_queue.

Referenced by [VulkanDevice\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.39.3.5 get\_physical\_device()

`void VulkanDevice::get_physical_device ( ) [private]`

Definition at line 72 of file [VulkanDevice.cpp](#).

```

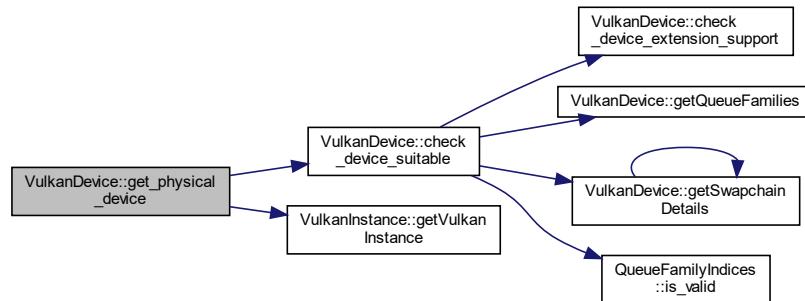
00072     {
00073         // Enumerate physical devices the vkInstance can access
00074         uint32_t device_count = 0;
00075         vkEnumeratePhysicalDevices(instance->getVulkanInstance(), &device_count,
00076                                     nullptr);
00077
00078         // if no devices available, then none support of Vulkan
00079         if (device_count == 0) {
00080             throw std::runtime_error(
00081                 "Can not find GPU's that support Vulkan Instance!");
00082         }
00083
00084         // Get list of physical devices
00085         std::vector<VkPhysicalDevice> device_list(device_count);
00086         vkEnumeratePhysicalDevices(instance->getVulkanInstance(), &device_count,
00087                                     device_list.data());
00088
00089         for (const auto& device : device_list) {
00090             if (check_device_suitable(device)) {
00091                 physical_device = device;
00092                 break;
00093             }
00094         }
00095
00096         // get properties of our new device
00097         vkGetPhysicalDeviceProperties(physical_device, &device_properties);
00098     }

```

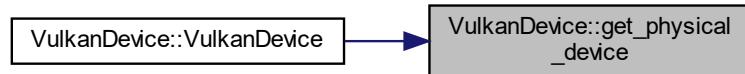
References [check\\_device\\_suitable\(\)](#), [device\\_properties](#), [VulkanInstance::getVulkanInstance\(\)](#), [instance](#), and [physical\\_device](#).

Referenced by [VulkanDevice\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.39.3.6 getComputeQueue()

VkQueue VulkanDevice::getComputeQueue () const [inline]

Definition at line 21 of file [VulkanDevice.hpp](#).

```
00021 { return compute_queue; };
```

References [compute\\_queue](#).

### 5.39.3.7 getGraphicsQueue()

VkQueue VulkanDevice::getGraphicsQueue () const [inline]

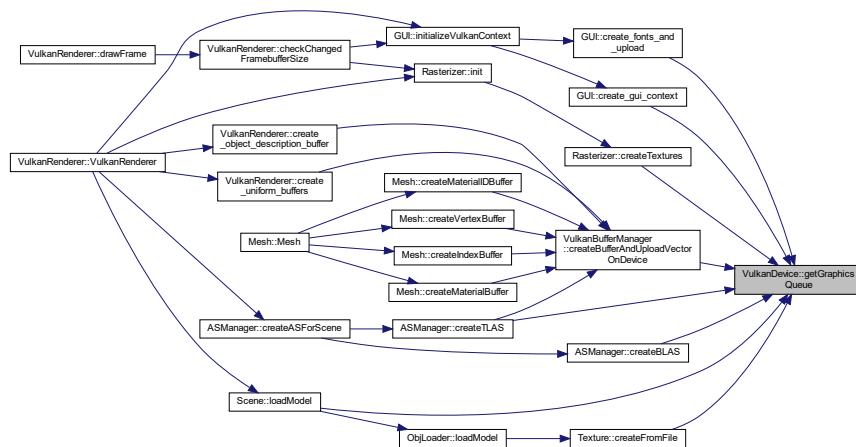
Definition at line 20 of file [VulkanDevice.hpp](#).

```
00020 { return graphics_queue; };
```

References [graphics\\_queue](#).

Referenced by [GUI::create\\_fonts\\_and\\_upload\(\)](#), [GUI::create\\_gui\\_context\(\)](#), [ASManager::createBLAS\(\)](#), [VulkanBufferManager::createTexture\(\)](#), [Rasterizer::createTextures\(\)](#), [ASManager::createTLAS\(\)](#), and [Scene::loadModel\(\)](#).

Here is the caller graph for this function:



### 5.39.3.8 getLogicalDevice()

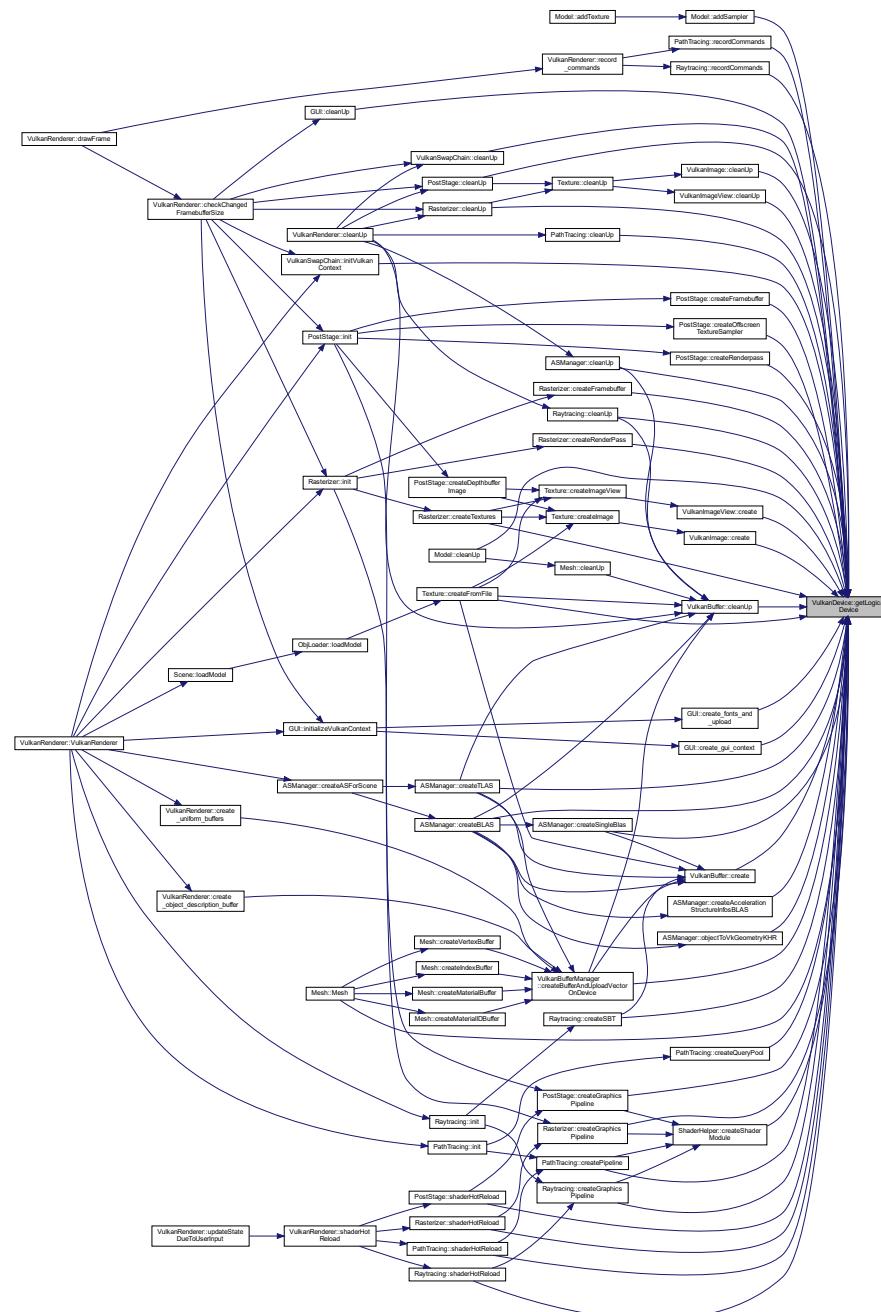
```
VkDevice VulkanDevice::getLogicalDevice ( ) const [inline]
```

Definition at line 18 of file [VulkanDevice.hpp](#).  
00018 { **return logical\_device;** };

References [logical\\_device](#).

Referenced by [Model::addSampler\(\)](#), [GUI::cleanUp\(\)](#), [ASManager::cleanUp\(\)](#), [PathTracing::cleanUp\(\)](#), [PostStage::cleanUp\(\)](#), [Rasterizer::cleanUp\(\)](#), [Raytracing::cleanUp\(\)](#), [Model::cleanUp\(\)](#), [VulkanBuffer::cleanUp\(\)](#), [VulkanImage::cleanUp\(\)](#), [VulkanImageView::cleanUp\(\)](#), [VulkanSwapChain::cleanUp\(\)](#), [VulkanImage::create\(\)](#), [VulkanImageView::create\(\)](#), [VulkanBuffer::create\(\)](#), [GUI::create\\_fonts\\_and\\_upload\(\)](#), [GUI::create\\_gui\\_context\(\)](#), [ASManager::createAccelerationStructureInfosBLAS\(\)](#), [ASManager::createBLAS\(\)](#), [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [PostStage::createFramebuffer\(\)](#), [Rasterizer::createFramebuffer\(\)](#), [Texture::createFromFile\(\)](#), [PostStage::createGraphicsPipeline\(\)](#), [Rasterizer::createGraphicsPipeline\(\)](#), [Raytracing::createGraphicsPipeline\(\)](#), [PostStage::createOffscreenTextureSampler\(\)](#), [PathTracing::createPipeline\(\)](#), [PathTracing::createQueryPool\(\)](#), [PostStage::createRenderpass\(\)](#), [Rasterizer::createRenderPass\(\)](#), [Raytracing::createSBT\(\)](#), [ShaderHelper::createShaderModule\(\)](#), [ASManager::createSingleBlas\(\)](#), [Rasterizer::createTextures\(\)](#), [ASManager::createTLAS\(\)](#), [VulkanSwapChain::initVulkanContext\(\)](#), [Mesh::Mesh\(\)](#), [ASManager::objectToVkGeometryKHR\(\)](#), [PathTracing::recordCommands\(\)](#), [Raytracing::recordCommands\(\)](#), [PathTracing::shaderHotReload\(\)](#), [PostStage::shaderHotReload\(\)](#), [Rasterizer::shaderHotReload\(\)](#), and [Raytracing::shaderHotReload\(\)](#).

Here is the caller graph for this function:



### 5.39.3.9 getPhysicalDevice()

```
VkPhysicalDevice VulkanDevice::getPhysicalDevice ( ) const [inline]
```

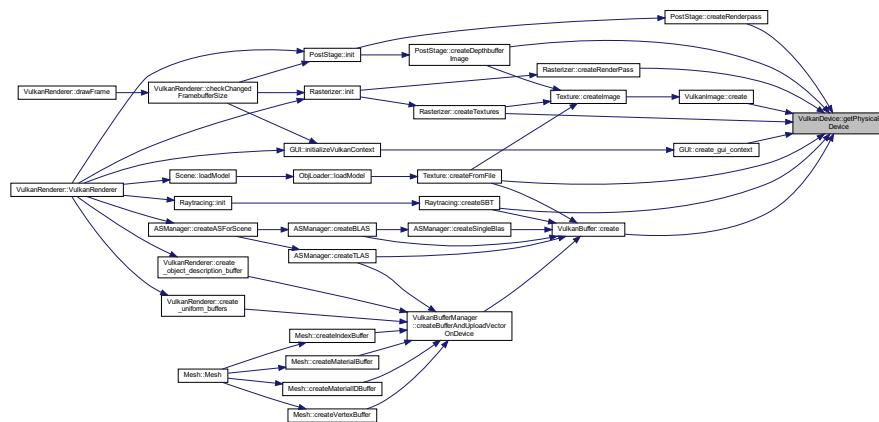
Definition at line 17 of file [VulkanDevice.hpp](#).

```
00017 { return physical_device; };
```

References [physical\\_device](#).

Referenced by [VulkanImage::create\(\)](#), [VulkanBuffer::create\(\)](#), [GUI::create\\_gui\\_context\(\)](#), [PostStage::createDepthbufferImage\(\)](#), [Texture::createFromFile\(\)](#), [PostStage::createRenderpass\(\)](#), [Rasterizer::createRenderPass\(\)](#), [Raytracing::createSBT\(\)](#), and [Rasterizer::createTextures\(\)](#).

Here is the caller graph for this function:



#### 5.39.3.10 getPhysicalDeviceProperties()

```
VkPhysicalDeviceProperties VulkanDevice::getPhysicalDeviceProperties () [inline]
```

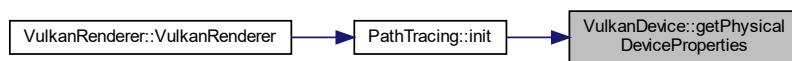
Definition at line 14 of file [VulkanDevice.hpp](#).

```
00014
00015     return device_properties;
00016 };
```

## References device properties.

Referenced by [PathTracing::init\(\)](#).

Here is the caller graph for this function:



#### 5.39.3.11 getPresentationQueue()

```
VkQueue VulkanDevice::getPresentationQueue ( ) const [inline]
```

Definition at line 22 of file VulkanDevice.hpp.

```
00022 { return presentation_queue; };
```

## References presentation\_queue.

### 5.39.3.12 `getQueueFamilies()` [1/2]

```
QueueFamilyIndices VulkanDevice::getQueueFamilies ( )
```

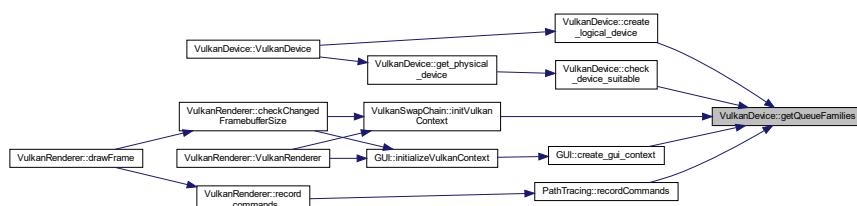
Definition at line 23 of file `VulkanDevice.cpp`.

```
00023 QueueFamilyIndices indices{};                                {  
00024     uint32_t queue_family_count = 0;  
00025     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,  
00026                                                 nullptr);  
00027  
00028     std::vector<VkQueueFamilyProperties> queue_family_list(queue_family_count);  
00029     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,  
00030                                                 queue_family_list.data());  
00031  
00032     // Go through each queue family and check if it has at least 1 of required  
00033     // types we need to keep track th eindex by our own  
00034     int index = 0;  
00035     for (const auto& queue_family : queue_family_list) {  
00036         // first check if queue family has at least 1 queue in that family  
00037         // Queue can be multiple types defined through bitfield. Need to bitwise AND  
00038         // with VK_QUE_*_BIT to check if has required type  
00039         if (queue_family.queueCount > 0 &&  
00040             queue_family.queueFlags & VK_QUEUE_GRAPHICS_BIT) {  
00041             indices.graphics_family = index; // if queue family valid, than get index  
00042         }  
00043  
00044         if (queue_family.queueCount > 0 &&  
00045             queue_family.queueFlags & VK_QUEUE_COMPUTE_BIT) {  
00046             indices.compute_family = index;  
00047         }  
00048  
00049         // check if queue family supports presentation  
00050         VkBool32 presentation_support = false;  
00051         vkGetPhysicalDeviceSurfaceSupportKHR(physical_device, index, *surface,  
00052                                                 &presentation_support);  
00053         // check if queue is presentation type (can be both graphics and  
00054         // presentation)  
00055         if (queue_family.queueCount > 0 && presentation_support) {  
00056             indices.presentation_family = index;  
00057         }  
00058  
00059         // check if queue family indices are in a valid state  
00060         if (indices.is_valid()) {  
00061             break;  
00062         }  
00063  
00064         index++;  
00065     }  
00066  
00067 }  
00068  
00069 return indices;  
00070 }
```

References `physical_device`, and `surface`.

Referenced by `check_device_suitable()`, `GUI::create_gui_context()`, `create_logical_device()`, `VulkanSwapChain::initVulkanContext()`, and `PathTracing::recordCommands()`.

Here is the caller graph for this function:



### 5.39.3.13 getQueueFamilies() [2/2]

```
QueueFamilyIndices VulkanDevice::getQueueFamilies (
    VkPhysicalDevice physical_device) [private]

Definition at line 239 of file VulkanDevice.cpp.
00240     {
00241     QueueFamilyIndices indices{};
00242
00243     uint32_t queue_family_count = 0;
00244     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
00245                                                 nullptr);
00246
00247     std::vector<VkQueueFamilyProperties> queue_family_list(queue_family_count);
00248     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
00249                                               queue_family_list.data());
00250
00251     // Go through each queue family and check if it has at least 1 of required
00252     // types we need to keep track th eindex by our own
00253     int index = 0;
00254     for (const auto& queue_family : queue_family_list) {
00255         // first check if queue family has at least 1 queue in that family
00256         // Queue can be multiple types defined through bitfield. Need to bitwise AND
00257         // with VK_QUEUE_*_BIT to check if has required type
00258         if (queue_family.queueCount > 0 &&
00259             queue_family.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
00260             indices.graphics_family = index; // if queue family valid, than get index
00261         }
00262
00263         if (queue_family.queueCount > 0 &&
00264             queue_family.queueFlags & VK_QUEUE_COMPUTE_BIT) {
00265             indices.compute_family = index;
00266         }
00267
00268         // check if queue family supports presentation
00269         VkBool32 presentation_support = false;
00270         vkGetPhysicalDeviceSurfaceSupportKHR(physical_device, index, *surface,
00271                                             &presentation_support);
00272         // check if queue is presentation type (can be both graphics and
00273         // presentation)
00274         if (queue_family.queueCount > 0 && presentation_support) {
00275             indices.presentation_family = index;
00276         }
00277
00278         // check if queue family indices are in a valid state
00279         if (!indices.is_valid()) {
00280             break;
00281         }
00282
00283         index++;
00284     }
00285
00286     return indices;
00287 }
```

References [physical\\_device](#), and [surface](#).

### 5.39.3.14 getSwapchainDetails() [1/2]

```
SwapChainDetails VulkanDevice::getSwapchainDetails ( )
```

**Definition at line 15 of file VulkanDevice.cpp.**

```
00015     {
00016     return getSwapchainDetails(physical_device);
00017 }
```

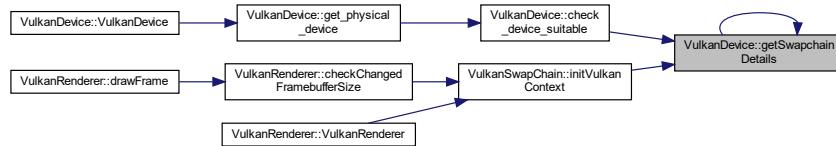
References [getSwapchainDetails\(\)](#), and [physical\\_device](#).

Referenced by [check\\_device\\_suitable\(\)](#), [getSwapchainDetails\(\)](#), and [VulkanSwapChain::initVulkanContext\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.39.3.15 getSwapchainDetails() [2/2]

```
SwapChainDetails VulkanDevice::getSwapchainDetails (
    VkPhysicalDevice device ) [private]
```

Definition at line 289 of file [VulkanDevice.cpp](#).

```

00289
00290     SwapChainDetails swapchain_details{};
00291     // get the surface capabilities for the given surface on the given physical
00292     // device
00293     vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
00294         device, *surface, &swapchain_details.surface_capabilities);
00295
00296     uint32_t format_count = 0;
00297     vkGetPhysicalDeviceSurfaceFormatsKHR(device, *surface, &format_count,
00298                                         nullptr);
00299
00300     // if formats returned, get list of formats
00301     if (format_count != 0) {
00302         swapchain_details.formats.resize(format_count);
00303         vkGetPhysicalDeviceSurfaceFormatsKHR(device, *surface, &format_count,
00304                                             swapchain_details.formats.data());
00305     }
00306
00307     uint32_t presentation_count = 0;
00308     vkGetPhysicalDeviceSurfacePresentModesKHR(device, *surface,
00309                                                 &presentation_count, nullptr);
00310
00311     // if presentation modes returned, get list of presentation modes
00312     if (presentation_count > 0) {
00313         swapchain_details.presentation_mode.resize(presentation_count);
00314         vkGetPhysicalDeviceSurfacePresentModesKHR(
00315             device, *surface, &presentation_count,
00316             swapchain_details.presentation_mode.data());
00317     }
00318
00319     return swapchain_details;
00320 }
```

References [surface](#).

## 5.39.4 Field Documentation

### 5.39.4.1 compute\_queue

```
VkQueue VulkanDevice::compute_queue [private]
```

Definition at line 41 of file [VulkanDevice.hpp](#).

Referenced by [create\\_logical\\_device\(\)](#), and [getComputeQueue\(\)](#).

### 5.39.4.2 device\_extensions

```
const std::vector<const char*> VulkanDevice::device_extensions [private]
```

#### Initial value:

```
= {  
    VK_KHR_SWAPCHAIN_EXTENSION_NAME  
}
```

Definition at line 52 of file [VulkanDevice.hpp](#).

Referenced by [check\\_device\\_extension\\_support\(\)](#), and [create\\_logical\\_device\(\)](#).

### 5.39.4.3 device\_extensions\_for\_raytracing

```
const std::vector<const char*> VulkanDevice::device_extensions_for_raytracing [private]
```

#### Initial value:

```
= {  
  
    VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME,  
    VK_KHR_RAY_TRACING_PIPELINE_EXTENSION_NAME,  
  
    VK_KHR_BUFFER_DEVICE_ADDRESS_EXTENSION_NAME,  
    VK_KHR_DEFERRED_HOST_OPERATIONS_EXTENSION_NAME,  
    VK_EXT_DESCRIPTOR_INDEXING_EXTENSION_NAME,  
  
    VK_KHR_SPIRV_1_4_EXTENSION_NAME,  
    VK_KHR_SHADER_FLOAT_CONTROLS_EXTENSION_NAME,  
    VK_KHR_PIPELINE_LIBRARY_EXTENSION_NAME,  
    VK_KHR_RAY_QUERY_EXTENSION_NAME  
}
```

Definition at line 59 of file [VulkanDevice.hpp](#).

Referenced by [create\\_logical\\_device\(\)](#).

#### 5.39.4.4 device\_properties

```
VkPhysicalDeviceProperties VulkanDevice::device_properties [private]
```

Definition at line 31 of file [VulkanDevice.hpp](#).

Referenced by [check\\_device\\_suitable\(\)](#), [get\\_physical\\_device\(\)](#), and [getPhysicalDeviceProperties\(\)](#).

#### 5.39.4.5 graphics\_queue

```
VkQueue VulkanDevice::graphics_queue [private]
```

Definition at line 39 of file [VulkanDevice.hpp](#).

Referenced by [create\\_logical\\_device\(\)](#), and [getGraphicsQueue\(\)](#).

#### 5.39.4.6 instance

```
VulkanInstance* VulkanDevice::instance [private]
```

Definition at line 35 of file [VulkanDevice.hpp](#).

Referenced by [get\\_physical\\_device\(\)](#), and [VulkanDevice\(\)](#).

#### 5.39.4.7 logical\_device

```
VkDevice VulkanDevice::logical_device [private]
```

Definition at line 33 of file [VulkanDevice.hpp](#).

Referenced by [cleanUp\(\)](#), [create\\_logical\\_device\(\)](#), and [getLogicalDevice\(\)](#).

#### 5.39.4.8 physical\_device

```
VkPhysicalDevice VulkanDevice::physical_device [private]
```

Definition at line 30 of file [VulkanDevice.hpp](#).

Referenced by [create\\_logical\\_device\(\)](#), [get\\_physical\\_device\(\)](#), [getPhysicalDevice\(\)](#), [getQueueFamilies\(\)](#), and [getSwapchainDetails\(\)](#).

### 5.39.4.9 presentation\_queue

```
VkQueue VulkanDevice::presentation_queue [private]
```

Definition at line 40 of file [VulkanDevice.hpp](#).

Referenced by [create\\_logical\\_device\(\)](#), and [getPresentationQueue\(\)](#).

### 5.39.4.10 surface

```
VkSurfaceKHR* VulkanDevice::surface [private]
```

Definition at line 36 of file [VulkanDevice.hpp](#).

Referenced by [getQueueFamilies\(\)](#), [getSwapchainDetails\(\)](#), and [VulkanDevice\(\)](#).

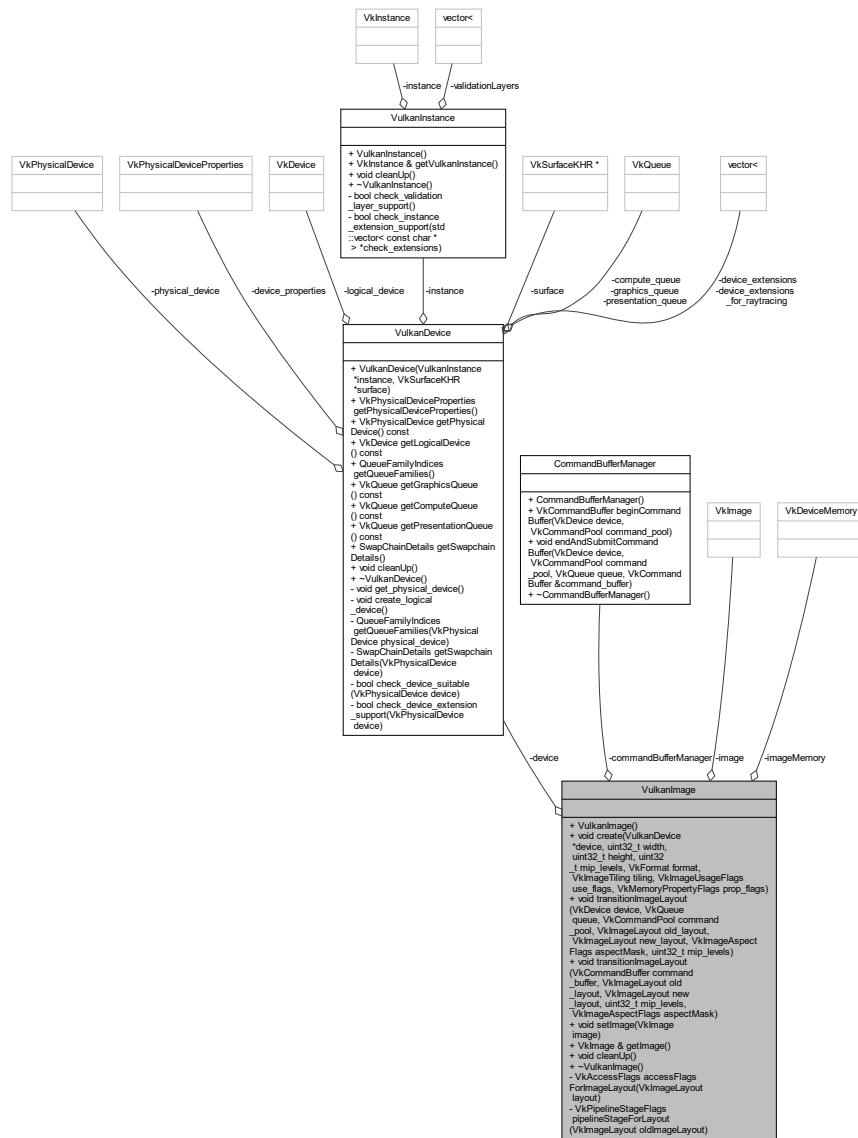
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/[VulkanDevice.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/[VulkanDevice.cpp](#)

## 5.40 VulkanImage Class Reference

```
#include <VulkanImage.hpp>
```

## Collaboration diagram for VulkanImage:



# Public Member Functions

- `VulkanImage ()`
  - `void create (VulkanDevice *device, uint32_t width, uint32_t height, uint32_t mip_levels, VkFormat format, VkImageTiling tiling, VkImageUsageFlags use_flags, VkMemoryPropertyFlags prop_flags)`
  - `void transitionImageLayout (VkDevice device, VkQueue queue, VkCommandPool command_pool, VkImageLayout old_layout, VkImageLayout new_layout, VkImageAspectFlags aspectMask, uint32_t mip_levels)`
  - `void transitionImageLayout (VkCommandBuffer command_buffer, VkImageLayout old_layout, VkImageLayout new_layout, uint32_t mip_levels, VkImageAspectFlags aspectMask)`
  - `void setImage (VklImage image)`
  - `VkImage & getImage ()`
  - `void cleanUp ()`
  - `~VulkanImage ()`

## Private Member Functions

- VkAccessFlags [accessFlagsForImageLayout](#) (VklImageLayout layout)
- VkPipelineStageFlags [pipelineStageForLayout](#) (VklImageLayout oldImageLayout)

## Private Attributes

- [VulkanDevice \\* device](#) {VK\_NULL\_HANDLE}
- [CommandBufferManager commandBufferManager](#)
- [VklImage image](#)
- [VkDeviceMemory imageMemory](#)

### 5.40.1 Detailed Description

Definition at line [7](#) of file [VulkanImage.hpp](#).

### 5.40.2 Constructor & Destructor Documentation

#### 5.40.2.1 [VulkanImage\(\)](#)

```
VulkanImage::VulkanImage ( )
```

Definition at line [6](#) of file [VulkanImage.cpp](#).  
00006 {}

#### 5.40.2.2 [~VulkanImage\(\)](#)

```
VulkanImage::~VulkanImage ( )
```

Definition at line [173](#) of file [VulkanImage.cpp](#).  
00173 {}

### 5.40.3 Member Function Documentation

### 5.40.3.1 accessFlagsForImageLayout()

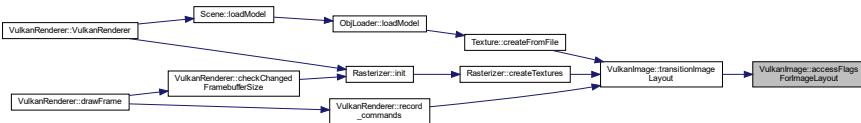
```
VkAccessFlags VulkanImage::accessFlagsForImageLayout (
    VkImageLayout layout ) [private]
```

Definition at line 175 of file [VulkanImage.cpp](#).

```
00175
00176     switch (layout) {
00177         case VK_IMAGE_LAYOUT_PREINITIALIZED:
00178             return VK_ACCESS_HOST_WRITE_BIT;
00179         case VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:
00180             return VK_ACCESS_TRANSFER_WRITE_BIT;
00181         case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
00182             return VK_ACCESS_TRANSFER_READ_BIT;
00183         case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
00184             return VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
00185         case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
00186             return VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
00187         case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
00188             return VK_ACCESS_SHADER_READ_BIT;
00189         default:
00190             return VkAccessFlags();
00191     }
00192 }
```

Referenced by [transitionImageLayout\(\)](#).

Here is the caller graph for this function:



### 5.40.3.2 cleanUp()

```
void VulkanImage::cleanUp ( )
```

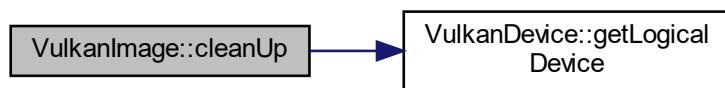
Definition at line 168 of file [VulkanImage.cpp](#).

```
00168     {
00169         vkDestroyImage(device->getLogicalDevice(), image, nullptr);
00170         vkFreeMemory(device->getLogicalDevice(), imageMemory, nullptr);
00171     }
```

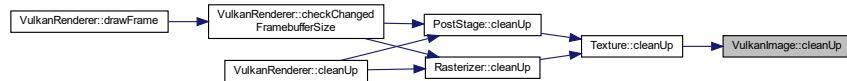
References `device`, `VulkanDevice::getLogicalDevice()`, `image`, and `imageMemory`.

Referenced by [Texture::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.40.3.3 create()

```

void VulkanImage::create (
    VulkanDevice * device,
    uint32_t width,
    uint32_t height,
    uint32_t mip_levels,
    VkFormat format,
    VkImageTiling tiling,
    VkImageUsageFlags use_flags,
    VkMemoryPropertyFlags prop_flags )
  
```

Definition at line 8 of file [VulkanImage.cpp](#).

```

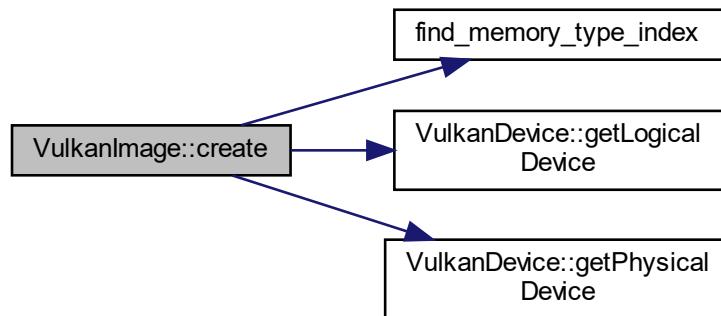
00011
00012     this->device = device;
00013     // CREATE image
00014     // image creation info
00015     VkImageCreateInfo image_create_info{};
00016     image_create_info.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
00017     image_create_info.imageType = VK_IMAGE_TYPE_2D; // type of image (1D, 2D, 3D)
00018     image_create_info.extent.width = width; // width if image extent
00019     image_create_info.extent.height = height; // height if image extent
00020     image_create_info.extent.depth = 1; // height if image extent
00021     image_create_info.mipLevels = mip_levels; // number of mipmap levels
00022     image_create_info.arrayLayers = 1; // number of levels in image array
00023     image_create_info.format = format; // format type of image
00024     image_create_info.tiling =
00025         tiling; // tiling of image ("arranged" for optimal reading)
00026     image_create_info.initialLayout =
00027         VK_IMAGE_LAYOUT_UNDEFINED; // layout of image data on creation
00028     image_create_info.usage =
00029         use_flags; // bit flags defining what image will be used for
00030     image_create_info.samples =
00031         VK_SAMPLE_COUNT_1_BIT; // number of samples for multisampling
00032     image_create_info.sharingMode =
00033         VK_SHARING_MODE_EXCLUSIVE; // whether image can be shared between queues
00034
00035     VkResult result = vkCreateImage(device->getLogicalDevice(),
00036                                     &image_create_info, nullptr, &image);
00037     ASSERT_VULKAN(result, "Failed to create an image!")
00038
00039     // CREATE memory for image
00040     // get memory requirements for a type of image
00041     VkMemoryRequirements memory_requirements;
00042     vkGetImageMemoryRequirements(device->getLogicalDevice(), image,
00043                                 &memory_requirements);
00044
00045     // allocate memory using image requirements and user defined properties
00046     VkMemoryAllocateInfo memory_alloc_info{};
00047     memory_alloc_info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
00048     memory_alloc_info.allocationSize = memory_requirements.size;
00049     memory_alloc_info.memoryTypeIndex =
00050         find_memory_type_index(device->getPhysicalDevice(),
00051                               memory_requirements.memoryTypeBits, prop_flags);
00052
00053     result = vkAllocateMemory(device->getLogicalDevice(), &memory_alloc_info,
00054                               nullptr, &imageMemory);
00055     ASSERT_VULKAN(result, "Failed to allocate memory!")
00056
00057     // connect memory to image
  
```

```
00058     vkBindImageMemory(device->getLogicalDevice(), image, imageMemory, 0);
00059 }
```

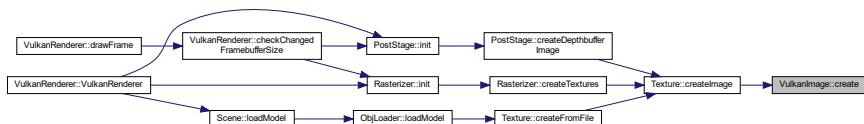
References [device](#), [find\\_memory\\_type\\_index\(\)](#), [VulkanDevice::getLogicalDevice\(\)](#), [VulkanDevice::getPhysicalDevice\(\)](#), [image](#), and [imageMemory](#).

Referenced by [Texture::createImage\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.40.3.4 getImage()

```
VkImage & VulkanImage::getImage( ) [inline]
```

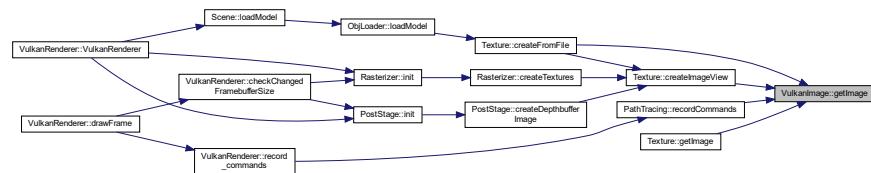
Definition at line 27 of file [VulkanImage.hpp](#).

```
00027 { return image; };
```

References [image](#).

Referenced by [Texture::createFromFile\(\)](#), [Texture::createImageView\(\)](#), [Texture::getImage\(\)](#), and [PathTracing::recordCommands\(\)](#).

Here is the caller graph for this function:



### 5.40.3.5 pipelineStageForLayout()

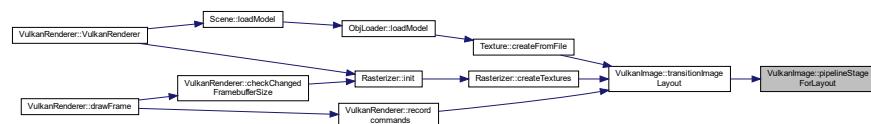
```
VkPipelineStageFlags VulkanImage::pipelineStageForLayout (
    VkImageLayout oldImageLayout ) [private]
```

Definition at line 194 of file [VulkanImage.cpp](#).

```
00195     {
00196         switch ( oldImageLayout ) {
00197             case VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:
00198             case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
00199                 return VK_PIPELINE_STAGE_TRANSFER_BIT;
00200             case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
00201                 return VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00202             case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
00203                 return VK_PIPELINE_STAGE_ALL_COMMANDS_BIT; // We do this to allow queue
00204                                         // other than graphic return
00205                                         // VK_PIPELINE_STAGE_early_fragment_tests_bit;
00206             case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
00207                 return VK_PIPELINE_STAGE_ALL_COMMANDS_BIT; // We do this to allow queue
00208                                         // other than graphic return
00209                                         // VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
00210             case VK_IMAGE_LAYOUT_PREINITIALIZED:
00211                 return VK_PIPELINE_STAGE_HOST_BIT;
00212             case VK_IMAGE_LAYOUT_UNDEFINED:
00213                 return VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
00214             default:
00215                 return VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
00216         }
00217     }
```

Referenced by [transitionImageLayout\(\)](#).

Here is the caller graph for this function:



### 5.40.3.6 setImage()

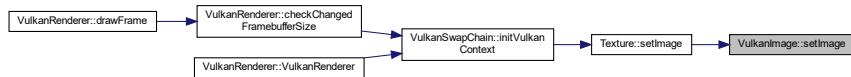
```
void VulkanImage::setImage (
    VkImage image )
```

Definition at line 166 of file [VulkanImage.cpp](#).  
 00166 { this->image = image; }

References [image](#).

Referenced by [Texture::setImage\(\)](#).

Here is the caller graph for this function:



### 5.40.3.7 transitionImageLayout() [1/2]

```
void VulkanImage::transitionImageLayout (
    VkCommandBuffer command_buffer,
    VkImageLayout old_layout,
    VkImageLayout new_layout,
    uint32_t mip_levels,
    VkImageAspectFlags aspectMask )
```

Definition at line 117 of file [VulkanImage.cpp](#).

```

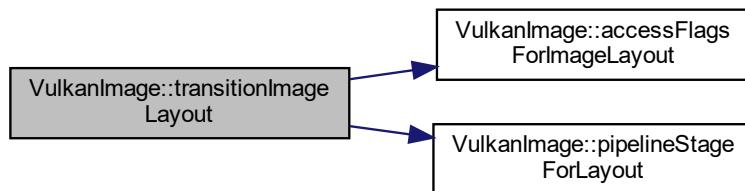
00121
00122     VkImageMemoryBarrier memory_barrier{};
00123     memory_barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00124     memory_barrier.oldLayout = old_layout;
00125     memory_barrier.newLayout = new_layout;
00126     memory_barrier.srcQueueFamilyIndex =
00127         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition from
00128     memory_barrier.dstQueueFamilyIndex =
00129         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition to
00130     memory_barrier.image =
00131         image; // image being accessed and modified as part of barrier
00132     memory_barrier.subresourceRange.aspectMask =
00133         aspectMask; // aspect of image being altered
00134     memory_barrier.subresourceRange.baseMipLevel =
00135         0; // first mip level to start alterations on
00136     memory_barrier.subresourceRange.levelCount =
00137         mip_levels; // number of mip levels to alter starting from baseMipLevel
00138     memory_barrier.subresourceRange.baseArrayLayer =
00139         0; // first layer to start alterations on
00140     memory_barrier.subresourceRange.layerCount =
00141         1; // number of layers to alter starting from baseArrayLayer
00142
00143     memory_barrier.srcAccessMask = accessFlagsForImageLayout(old_layout);
00144     memory_barrier.dstAccessMask = accessFlagsForImageLayout(new_layout);
00145
00146     VkPipelineStageFlags src_stage = pipelineStageForLayout(old_layout);
00147     VkPipelineStageFlags dst_stage = pipelineStageForLayout(new_layout);
00148
00149     // if transitioning from new image to image ready to receive data
00150
00151     vkCmdPipelineBarrier(
00152         command_buffer, src_stage,
00153         dst_stage, // pipeline stages (match to src and dst accessmask)
  
```

```

00155     0,           // no dependency flags
00156     0,
00157     nullptr,    // memory barrier count + data
00158     0,
00159     nullptr,    // buffer memory barrier count + data
00160     1,
00161     &memory_barrier // image memory barrier count + data
00162
00163 };
00164 }
```

References [accessFlagsForImageLayout\(\)](#), [image](#), and [pipelineStageForLayout\(\)](#).

Here is the call graph for this function:



#### 5.40.3.8 transitionImageLayout() [2/2]

```

void VulkanImage::transitionImageLayout (
    VkDevice device,
    VkQueue queue,
    VkCommandPool command_pool,
    VkImageLayout old_layout,
    VkImageLayout new_layout,
    VkImageAspectFlags aspectMask,
    uint32_t mip_levels )
```

Definition at line 61 of file [VulkanImage.cpp](#).

```

00066
00067     VkCommandBuffer command_buffer =
00068         commandBufferManager.beginCommandBuffer(device, command_pool);
00069
00070     // VK_IMAGE_ASPECT_COLOR_BIT
00071     VkImageMemoryBarrier memory_barrier{};
00072     memory_barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00073     memory_barrier.oldLayout = old_layout;
00074     memory_barrier.newLayout = new_layout;
00075     memory_barrier.srcQueueFamilyIndex =
00076         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition from
00077     memory_barrier.dstQueueFamilyIndex =
00078         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition to
00079     memory_barrier.image =
00080         image; // image being accessed and modified as part of barrier
00081     memory_barrier.subresourceRange.aspectMask =
00082         aspectMask; // aspect of image being altered
00083     memory_barrier.subresourceRange.baseMipLevel =
00084         0; // first mip level to start alterations on
00085     memory_barrier.subresourceRange.levelCount =
00086         mip_levels; // number of mip levels to alter starting from baseMipLevel
00087     memory_barrier.subresourceRange.baseArrayLayer =
00088         0; // first layer to start alterations on
00089     memory_barrier.subresourceRange.layerCount =
```

```

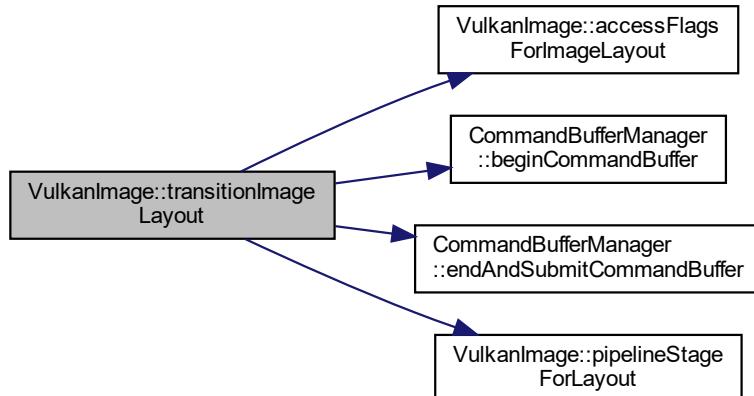
00090     1; // number of layers to alter starting from baseArrayLayer
00091
00092     // if transitioning from new image to image ready to receive data
00093     memory_barrier.srcAccessMask = accessFlagsForImageLayout(old_layout);
00094     memory_barrier.dstAccessMask = accessFlagsForImageLayout(new_layout);
00095
00096     VkPipelineStageFlags src_stage = pipelineStageForLayout(old_layout);
00097     VkPipelineStageFlags dst_stage = pipelineStageForLayout(new_layout);
00098
00099     vkCmdPipelineBarrier(
00100
00101         command_buffer, src_stage,
00102         dst_stage, // pipeline stages (match to src and dst accessmask)
00103         0,           // no dependency flags
00104         0,
00105         nullptr, // memory barrier count + data
00106         0,
00107         nullptr, // buffer memory barrier count + data
00108         1,
00109         &memory_barrier // image memory barrier count + data
00110
00111     );
00112
00113     commandBufferManager.endAndSubmitCommandBuffer(device, command_pool, queue,
00114                                                 command_buffer);
00115 }

```

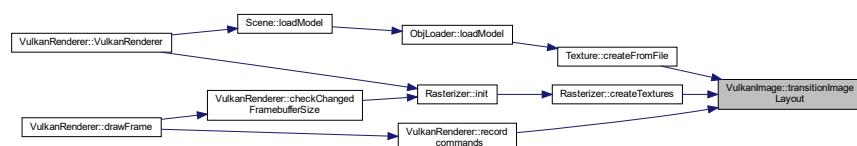
References [accessFlagsForImageLayout\(\)](#), [CommandBufferManager::beginCommandBuffer\(\)](#), [commandBufferManager](#), [device](#), [CommandBufferManager::endAndSubmitCommandBuffer\(\)](#), [image](#), and [pipelineStageForLayout\(\)](#).

Referenced by [Texture::createFromFile\(\)](#), [Rasterizer::createTextures\(\)](#), and [VulkanRenderer::record\\_commands\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.40.4 Field Documentation

### 5.40.4.1 commandBufferManager

```
CommandBufferManager VulkanImage::commandBufferManager [private]
```

Definition at line 35 of file [VulkanImage.hpp](#).

Referenced by [transitionImageLayout\(\)](#).

### 5.40.4.2 device

```
VulkanDevice* VulkanImage::device {VK_NULL_HANDLE} [private]
```

Definition at line 34 of file [VulkanImage.hpp](#).

Referenced by [cleanUp\(\)](#), [create\(\)](#), and [transitionImageLayout\(\)](#).

### 5.40.4.3 image

```
VkImage VulkanImage::image [private]
```

Definition at line 37 of file [VulkanImage.hpp](#).

Referenced by [cleanUp\(\)](#), [create\(\)](#), [getImage\(\)](#), [setImage\(\)](#), and [transitionImageLayout\(\)](#).

### 5.40.4.4 imageMemory

```
VkDeviceMemory VulkanImage::imageMemory [private]
```

Definition at line 38 of file [VulkanImage.hpp](#).

Referenced by [cleanUp\(\)](#), and [create\(\)](#).

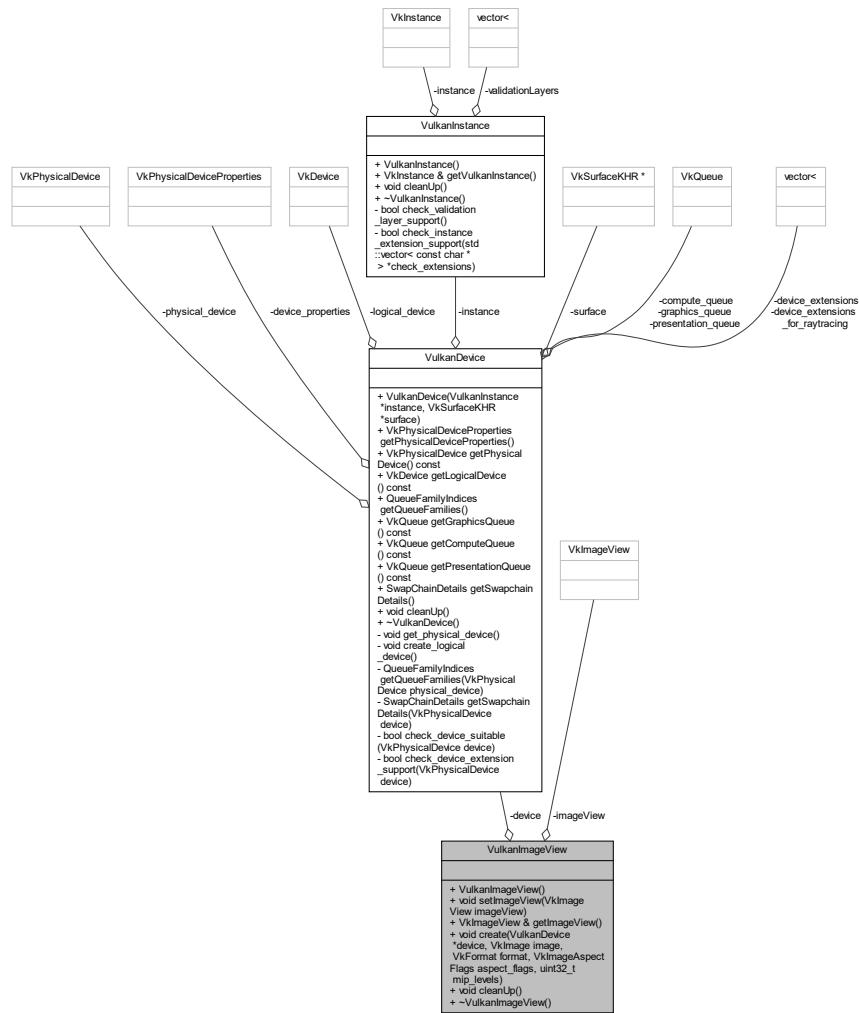
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/[VulkanImage.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/[VulkanImage.cpp](#)

## 5.41 VulkanImageView Class Reference

```
#include <VulkanImageView.hpp>
```

Collaboration diagram for VulkanImageView:



### Public Member Functions

- [VulkanImageView \(\)](#)
- void [setImageView \(VkImage imageView\)](#)
- [VkImage & getImageView \(\)](#)
- void [create \(VulkanDevice \\*device, VkImage image, VkFormat format, VkImageAspectFlags aspect\\_flags, uint32\\_t mip\\_levels\)](#)
- void [cleanUp \(\)](#)
- [~VulkanImageView \(\)](#)

### Private Attributes

- [VulkanDevice \\* device {VK\\_NULL\\_HANDLE}](#)
- [VkImage imageView](#)

### 5.41.1 Detailed Description

Definition at line 6 of file [VulkanImageView.hpp](#).

### 5.41.2 Constructor & Destructor Documentation

#### 5.41.2.1 VulkanImageView()

```
VulkanImageView::VulkanImageView ( )
```

Definition at line 3 of file [VulkanImageView.cpp](#).  
00003 { }

#### 5.41.2.2 ~VulkanImageView()

```
VulkanImageView::~VulkanImageView ( )
```

Definition at line 49 of file [VulkanImageView.cpp](#).  
00049 { }

### 5.41.3 Member Function Documentation

#### 5.41.3.1 cleanUp()

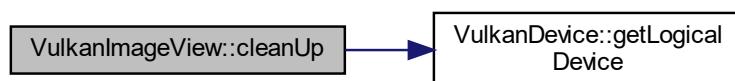
```
void VulkanImageView::cleanUp ( )
```

Definition at line 45 of file [VulkanImageView.cpp](#).  
00045 {  
00046 vkDestroyImageView(device->[getLogicalDevice\(\)](#), imageView, nullptr);  
00047 }

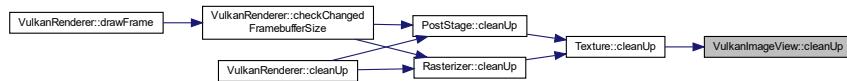
References [device](#), [VulkanDevice::getLogicalDevice\(\)](#), and [imageView](#).

Referenced by [Texture::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.41.3.2 create()

```
void VulkanImageView::create (
    VulkanDevice * device,
    VkImage image,
    VkFormat format,
    VkImageAspectFlags aspect_flags,
    uint32_t mip_levels )
```

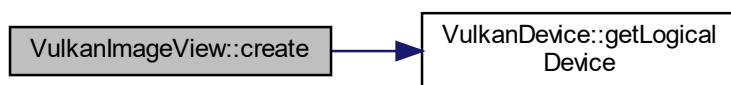
Definition at line 9 of file [VulkanImageView.cpp](#).

```
00011     this->device = device;
00012
00013
00014     VkImageViewCreateInfo view_create_info{};
00015     view_create_info.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
00016     view_create_info.image = image; // image to create view for
00017     view_create_info.viewType = VK_IMAGE_VIEW_TYPE_2D; // typ of image
00018     view_create_info.format = format;
00019     view_create_info.components.r =
00020         VK_COMPONENT_SWIZZLE_IDENTITY; // allows remapping of rgba components to
00021         // other rgba values
00022     view_create_info.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
00023     view_create_info.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
00024     view_create_info.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
00025
00026     // subresources allow the view to view only a part of an image
00027     view_create_info.subresourceRange.aspectMask =
00028         aspect_flags; // which aspect of an image to view (e.g. color bit for
00029         // viewing color)
00030     view_create_info.subresourceRange.baseMipLevel =
00031         0; // start mipmap level to view from
00032     view_create_info.subresourceRange.levelCount =
00033         mip_levels; // number of mipmap levels to view
00034     view_create_info.subresourceRange.baseArrayLayer =
00035         0; // start array level to view from
00036     view_create_info.subresourceRange.layerCount =
00037         1; // number of array levels to view
00038
00039     // create image view
00040     VkResult result = vkCreateImageView(device->getLogicalDevice(),
00041                                         &view_create_info, nullptr, &imageView);
00042     ASSERT_VULKAN(result, "Failed to create an image view!")
00043 }
```

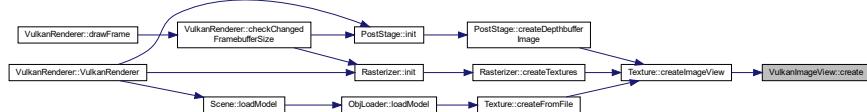
References [device](#), [VulkanDevice::getLogicalDevice\(\)](#), and [imageView](#).

Referenced by [Texture::createImageView\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.41.3.3 getImageView()

`VkImageView & VulkanImageView::getImageView () [inline]`

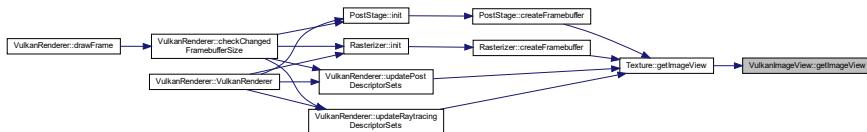
Definition at line 12 of file [VulkanImageView.hpp](#).

```
00012 { return imageView; };
```

References [imageView](#).

Referenced by [Texture::getImageView\(\)](#).

Here is the caller graph for this function:



### 5.41.3.4 setImageView()

```
void VulkanImageView::setImageView (
    VkImageView imageView )
```

Definition at line 5 of file [VulkanImageView.cpp](#).

```
00005 {
00006     this->imageView = imageView;
00007 }
```

References [imageView](#).

Referenced by [Texture::setImageView\(\)](#).

Here is the caller graph for this function:



## 5.41.4 Field Documentation

### 5.41.4.1 device

```
VulkanDevice* VulkanImageView::device {VK_NULL_HANDLE} [private]
```

Definition at line 22 of file [VulkanImageView.hpp](#).

Referenced by [cleanUp\(\)](#), and [create\(\)](#).

### 5.41.4.2 imageView

```
VkImageView VulkanImageView::imageView [private]
```

Definition at line 24 of file [VulkanImageView.hpp](#).

Referenced by [cleanUp\(\)](#), [create\(\)](#), [getImageView\(\)](#), and [setImageView\(\)](#).

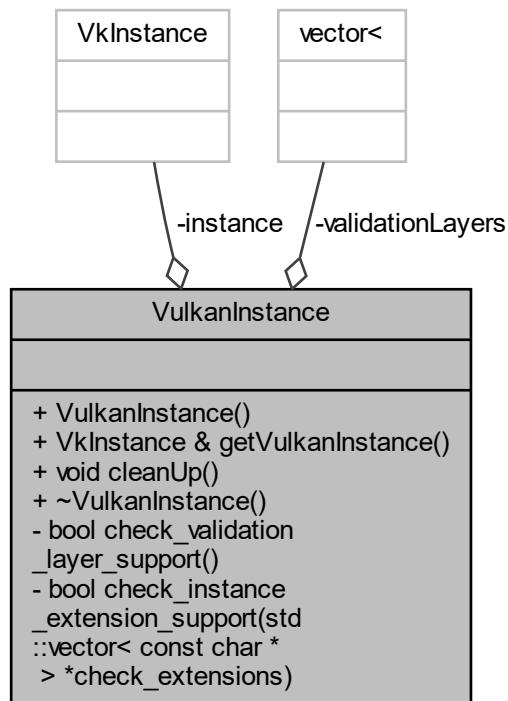
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/[VulkanImageView.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/[VulkanImageView.cpp](#)

## 5.42 VulkanInstance Class Reference

```
#include <VulkanInstance.hpp>
```

Collaboration diagram for VulkanInstance:



## Public Member Functions

- [VulkanInstance \(\)](#)
- [VkInstance & getVulkanInstance \(\)](#)
- [void cleanUp \(\)](#)
- [~VulkanInstance \(\)](#)

## Private Member Functions

- [bool check\\_validation\\_layer\\_support \(\)](#)
- [bool check\\_instance\\_extension\\_support \(std::vector<const char \\* > \\*check\\_extensions\)](#)

## Private Attributes

- [VkInstance instance](#)
- [std::vector<const char \\* > validationLayers = {"VK\\_LAYER\\_KHRONOS\\_validation"}](#)

### 5.42.1 Detailed Description

Definition at line 11 of file [VulkanInstance.hpp](#).

## 5.42.2 Constructor & Destructor Documentation

### 5.42.2.1 VulkanInstance()

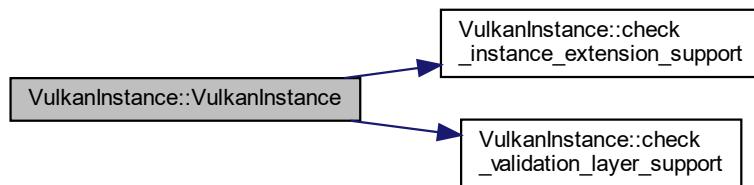
```
VulkanInstance::VulkanInstance ( )
```

Definition at line 7 of file [VulkanInstance.cpp](#).

```
00007             {
00008     if (ENABLE_VALIDATION_LAYERS && !check_validation_layer_support()) {
00009         throw std::runtime_error("Validation layers requested, but not available!");
00010     }
00011
00012     // info about app
00013     // most data doesn't affect program; is for developer convenience
00014     VkApplicationInfo app_info{};
00015     app_info.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
00016     app_info.pApplicationName =
00017         "\\\\_ Epic Graphics from hell \\\_"; // custom name of app
00018     app_info.applicationVersion =
00019         VK_MAKE_VERSION(1, 3, 1); // custom version of app
00020     app_info.pEngineName = "Cataglyphis Renderer"; // custom engine name
00021     app_info.engineVersion = VK_MAKE_VERSION(1, 3, 3); // custom engine version
00022     app_info.apiVersion = VK_API_VERSION_1_3; // the vulkan version
00023
00024     // creation info for a VkInstance
00025     VkInstanceCreateInfo create_info{};
00026     create_info.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
00027     create_info.pApplicationInfo = &app_info;
00028
00029     // add validation layers IF enabled to the create info struct
00030     if (ENABLE_VALIDATION_LAYERS) {
00031         create_info.enabledLayerCount =
00032             static_cast<uint32_t>(validationLayers.size());
00033         create_info.ppEnabledLayerNames = validationLayers.data();
00034
00035     } else {
00036         create_info.enabledLayerCount = 0;
00037         create_info.pNext = nullptr;
00038     }
00039
00040     // create list to hold instance extensions
00041     std::vector<const char*> instance_extensions = std::vector<const char*>();
00042
00043     // Setup extensions the instance will use
00044     uint32_t glfw_extensions_count = 0; // GLFW may require multiple extensions
00045     const char** glfw_extensions; // Extensions passed as array of cstrings, so
00046                                // need pointer(array) to pointer
00047
00048     // set GLFW extensions
00049     glfw_extensions = glfwGetRequiredInstanceExtensions(&glfw_extensions_count);
00050
00051     // Add GLFW extensions to list of extensions
00052     for (size_t i = 0; i < glfw_extensions_count; i++) {
00053         instance_extensions.push_back(glfw_extensions[i]);
00054     }
00055
00056     if (ENABLE_VALIDATION_LAYERS) {
00057         instance_extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
00058     }
00059
00060     // check instance extensions supported
00061     if (!check_instance_extension_support(&instance_extensions)) {
00062         throw std::runtime_error(
00063             "VkInstance does not support required extensions!");
00064     }
00065
00066     create_info.enabledExtensionCount =
00067         static_cast<uint32_t>(instance_extensions.size());
00068     create_info.ppEnabledExtensionNames = instance_extensions.data();
00069
00070     // create instance
00071     VkResult result = vkCreateInstance(&create_info, nullptr, &instance);
00072     ASSERT_VULKAN(result, "Failed to create a Vulkan instance!");
00073 }
```

References [check\\_instance\\_extension\\_support\(\)](#), [check\\_validation\\_layer\\_support\(\)](#), [ENABLE\\_VALIDATION\\_LAYERS](#), [instance](#), and [validationLayers](#).

Here is the call graph for this function:



#### 5.42.2.2 ~VulkanInstance()

VulkanInstance::~VulkanInstance ( )

Definition at line 133 of file [VulkanInstance.cpp](#).  
00133 { }

#### 5.42.3 Member Function Documentation

##### 5.42.3.1 check\_instance\_extension\_support()

```
bool VulkanInstance::check_instance_extension_support (
    std::vector< const char * > * check_extensions ) [private]
```

Definition at line 100 of file [VulkanInstance.cpp](#).

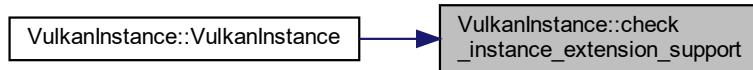
```

00101
00102 // Need to get number of extensions to create array of correct size to hold
00103 // extensions
00104 uint32_t extension_count = 0;
00105 vkEnumerateInstanceExtensionProperties(nullptr, &extension_count, nullptr);
00106
00107 // create a list of VkExtensionProperties using count
00108 std::vector<VkExtensionProperties> extensions(extension_count);
00109 vkEnumerateInstanceExtensionProperties(nullptr, &extension_count,
00110                                         extensions.data());
00111
00112 // check if given extensions are in list of available extensions
00113 for (const auto& check_extension : *check_extensions) {
00114     bool has_extension = false;
00115
00116     for (const auto& extension : extensions) {
00117         if (strcmp(check_extension, extension.extensionName)) {
00118             has_extension = true;
00119             break;
00120         }
00121     }
00122
00123     if (!has_extension) {
00124         return false;
00125     }
00126 }
00127 }
```

```
00128     return true;
00129 }
```

Referenced by [VulkanInstance\(\)](#).

Here is the caller graph for this function:



#### 5.42.3.2 check\_validation\_layer\_support()

```
bool VulkanInstance::check_validation_layer_support () [private]
```

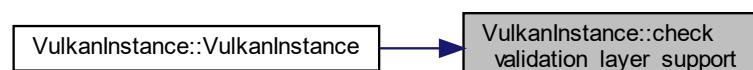
Definition at line 75 of file [VulkanInstance.cpp](#).

```
00075 {
00076     uint32_t layerCount;
00077     vkEnumerateInstanceLayerProperties(&layerCount, nullptr);
00078
00079     std::vector<VkLayerProperties> availableLayers(layerCount);
00080     vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());
00081
00082     for (const char* layerName : validationLayers) {
00083         bool layerFound = false;
00084
00085         for (const auto& layerProperties : availableLayers) {
00086             if (strcmp(layerName, layerProperties.layerName) == 0) {
00087                 layerFound = true;
00088                 break;
00089             }
00090         }
00091
00092         if (!layerFound) {
00093             return false;
00094         }
00095     }
00096
00097     return true;
00098 }
```

References [validationLayers](#).

Referenced by [VulkanInstance\(\)](#).

Here is the caller graph for this function:



### 5.42.3.3 cleanUp()

```
void VulkanInstance::cleanUp( )
```

Definition at line 131 of file [VulkanInstance.cpp](#).  
 00131 { vkDestroyInstance(instance, nullptr); }

References [instance](#).

Referenced by [VulkanRenderer::cleanUp\(\)](#).

Here is the caller graph for this function:



### 5.42.3.4 getVulkanInstance()

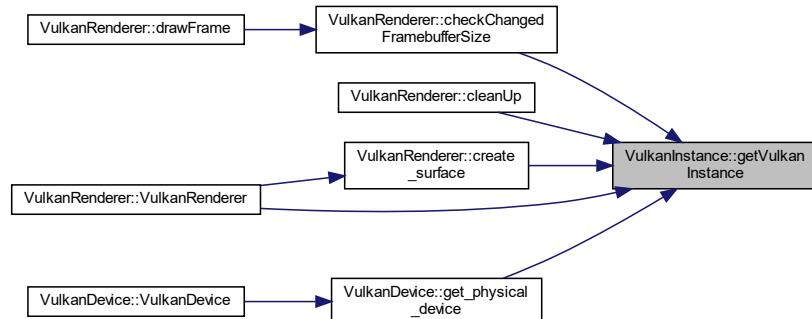
```
VkInstance & VulkanInstance::getVulkanInstance( ) [inline]
```

Definition at line 15 of file [VulkanInstance.hpp](#).  
 00015 { return instance; };

References [instance](#).

Referenced by [VulkanRenderer::checkChangedFramebufferSize\(\)](#), [VulkanRenderer::cleanUp\(\)](#), [VulkanRenderer::create\\_surface\(\)](#), [VulkanDevice::get\\_physical\\_device\(\)](#), and [VulkanRenderer::VulkanRenderer\(\)](#).

Here is the caller graph for this function:



#### **5.42.4 Field Documentation**

#### 5.42.4.1 instance

```
VkInstance VulkanInstance::instance [private]
```

Definition at line 22 of file [VulkanInstance.hpp](#).

Referenced by [cleanUp\(\)](#), [getVulkanInstance\(\)](#), and [VulkanInstance\(\)](#).

#### 5.42.4.2 validationLayers

```
std::vector<const char*> VulkanInstance::validationLayers = {"VK_LAYER_KHRONOS_validation"}  
[private]
```

Definition at line 25 of file [VulkanInstance.hpp](#).

Referenced by [check\\_validation\\_layer\\_support\(\)](#), and [VulkanInstance\(\)](#).

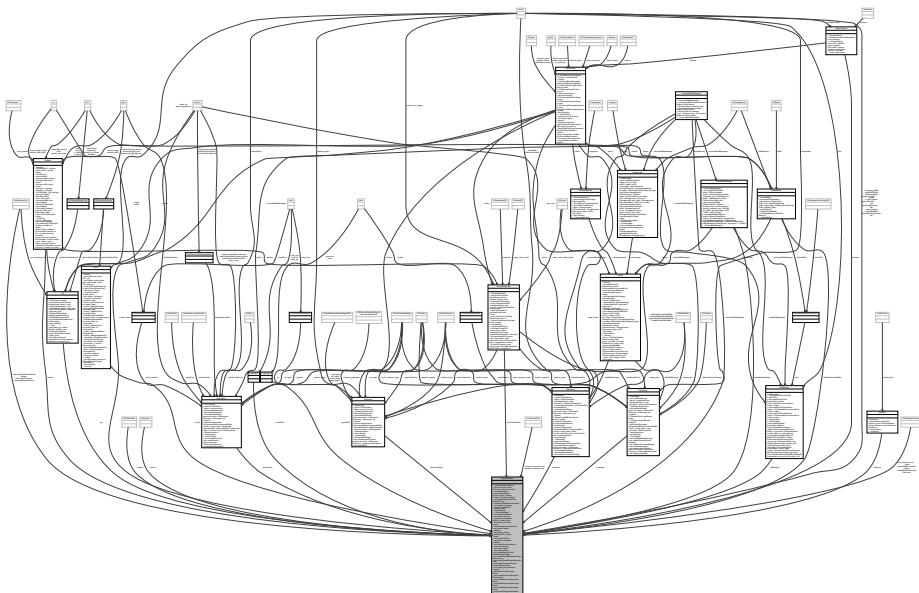
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/VulkanInstance.hpp
  - C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanInstance.cpp

## 5.43 VulkanRenderer Class Reference

```
#include <VulkanRenderer.hpp>
```

## Collaboration diagram for VulkanRenderer:



## Public Member Functions

- `VulkanRenderer (Window *window, Scene *scene, GUI *gui, Camera *camera)`
- `void drawFrame ()`
- `void updateUniforms (Scene *scene, Camera *camera, Window *window)`
- `void updateStateDueToUserInput (GUI *gui)`
- `void finishAllRenderCommands ()`
- `void update_raytracing_descriptor_set (uint32_t image_index)`
- `void cleanUp ()`
- `~VulkanRenderer ()`

## Private Member Functions

- `void shaderHotReload ()`
- `void create_surface ()`
- `void record_commands (uint32_t image_index)`
- `void create_command_pool ()`
- `void cleanUpCommandPools ()`
- `void create_uniform_buffers ()`
- `void update_uniform_buffers (uint32_t image_index)`
- `void cleanUpUBOs ()`
- `void create_command_buffers ()`
- `void createSynchronization ()`
- `void cleanUpSync ()`
- `void create_object_description_buffer ()`
- `void createDescriptorPoolSharedRenderStages ()`
- `void createSharedRenderDescriptorSetLayouts ()`
- `void createSharedRenderDescriptorSet ()`
- `void updateTexturesInSharedRenderDescriptorSet ()`
- `void create_post_descriptor_layout ()`
- `void updatePostDescriptorSets ()`
- `void createRaytracingDescriptorSetLayouts ()`
- `void createRaytracingDescriptorSets ()`
- `void updateRaytracingDescriptorSets ()`
- `void createRaytracingDescriptorPool ()`
- `bool checkChangedFramebufferSize ()`

## Private Attributes

- `VulkanBufferManager vulkanBufferManager`
- `VulkanInstance instance`
- `VkSurfaceKHR surface`
- `std::unique_ptr< VulkanDevice > device`
- `VulkanSwapChain vulkanSwapChain`
- `Window * window`
- `Scene * scene`
- `GUI * gui`
- `VkCommandPool graphics_command_pool`
- `VkCommandPool compute_command_pool`
- `GlobalUBO globalUBO`
- `std::vector< VulkanBuffer > globalUBOBuffer`
- `SceneUBO sceneUBO`
- `std::vector< VulkanBuffer > sceneUBOBuffer`

- std::vector< VkCommandBuffer > [command\\_buffers](#)
- CommandBufferManager [commandBufferManager](#)
- Raytracing [raytracingStage](#)
- Rasterizer [rasterizer](#)
- PathTracing [pathTracing](#)
- PostStage [postStage](#)
- Allocator [allocator](#)
- uint32\_t [current\\_frame](#) {0}
- std::vector< VkSemaphore > [image\\_available](#)
- std::vector< VkSemaphore > [render\\_finished](#)
- std::vector< VkFence > [in\\_flight\\_fences](#)
- std::vector< VkFence > [images\\_in\\_flight\\_fences](#)
- ASManager [asManager](#)
- VulkanBuffer [objectDescriptionBuffer](#)
- VkDescriptorPool [descriptorPoolSharedRenderStages](#)
- VkDescriptorSetLayout [sharedRenderDescriptorSetLayout](#)
- std::vector< VkDescriptorSet > [sharedRenderDescriptorSet](#)
- VkDescriptorPool [post\\_descriptor\\_pool](#) {VK\_NULL\_HANDLE}
- VkDescriptorSetLayout [post\\_descriptor\\_set\\_layout](#) {VK\_NULL\_HANDLE}
- std::vector< VkDescriptorSet > [post\\_descriptor\\_set](#)
- VkDescriptorPool [raytracingDescriptorPool](#) {VK\_NULL\_HANDLE}
- std::vector< VkDescriptorSet > [raytracingDescriptorSet](#)
- VkDescriptorSetLayout [raytracingDescriptorsetLayout](#) {VK\_NULL\_HANDLE}

### 5.43.1 Detailed Description

Definition at line [46](#) of file [VulkanRenderer.hpp](#).

### 5.43.2 Constructor & Destructor Documentation

#### 5.43.2.1 VulkanRenderer()

```
VulkanRenderer::VulkanRenderer (
    Window * window,
    Scene * scene,
    GUI * gui,
    Camera * camera )
```

Definition at line [24](#) of file [VulkanRenderer.cpp](#).

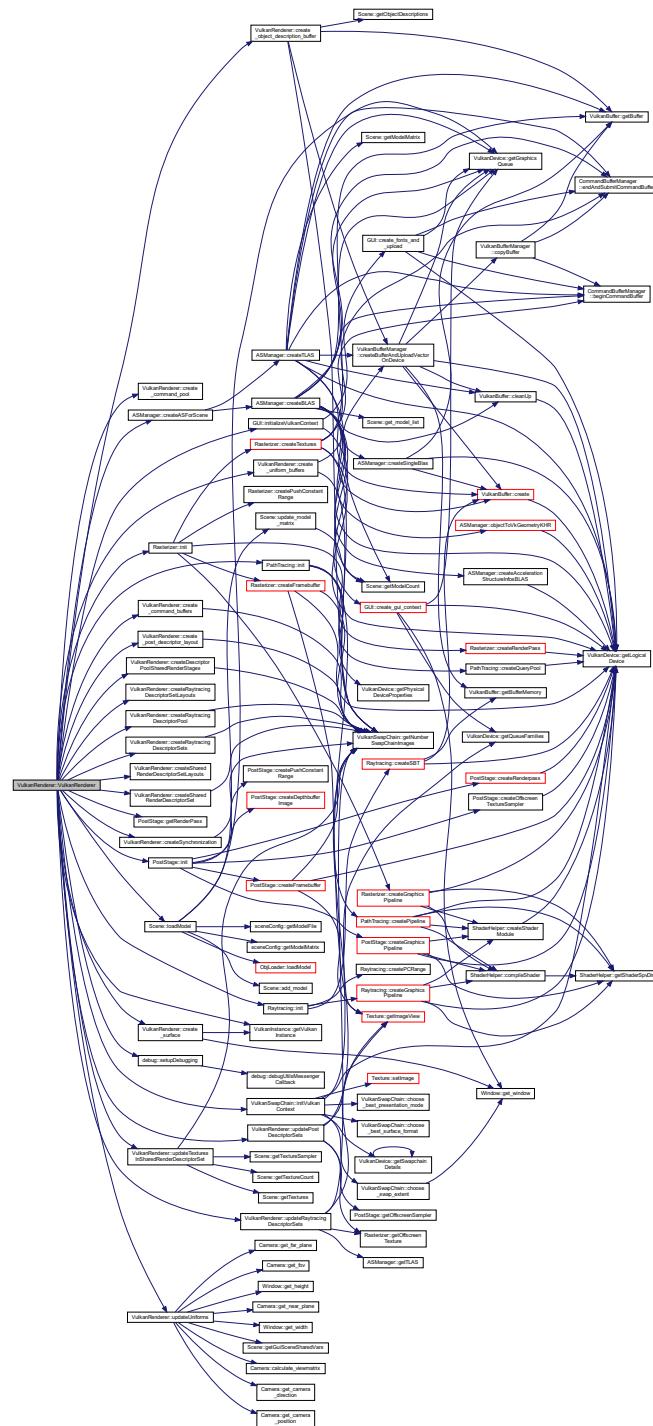
```
00026   :
00027
00028     window(window),
00029     scene(scene),
00030     gui(gui)
00031
00032 {
00033     updateUniforms(scene, camera, window);
00034
00035     try {
00036         instance = VulkanInstance();
00037
00038         VkDebugReportFlagsEXT debugReportFlags =
00039             VK_DEBUG_REPORT_ERROR_BIT_EXT | VK_DEBUG_REPORT_WARNING_BIT_EXT;
00040         if (ENABLE_VALIDATION_LAYERS)
```

```

00041     debug::setupDebugging(instance.getVulkanInstance(), debugReportFlags,
00042                             VK_NULL_HANDLE);
00043
00044     create_surface();
00045
00046     device = std::make_unique<VulkanDevice>(&instance, &surface);
00047
00048     allocator =
00049         Allocator(device->getLogicalDevice(), device->getPhysicalDevice(),
00050                     instance.getVulkanInstance());
00051
00052     create_command_pool();
00053
00054     vulkanSwapChain.initVulkanContext(device.get(), window, surface);
00055     create_uniform_buffers();
00056     create_command_buffers();
00057
00058     createSynchronization();
00059
00060     createSharedRenderDescriptorSetLayouts();
00061     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_rasterizer = {
00062         sharedRenderDescriptorSetLayout};
00063     rasterizer.init(device.get(), &vulkanSwapChain,
00064                     descriptor_set_layouts_rasterizer, graphics_command_pool);
00065     create_post_descriptor_layout();
00066     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
00067         post_descriptor_set_layout};
00068     postStage.init(device.get(), &vulkanSwapChain, descriptor_set_layouts_post);
00069     createDescriptorPoolSharedRenderStages();
00070     createSharedRenderDescriptorSet();
00071
00072     updatePostDescriptorSets();
00073
00074     createRaytracingDescriptorPool();
00075     createRaytracingDescriptorSetLayouts();
00076     std::vector<VkDescriptorSetLayout> layouts;
00077     layouts.push_back(sharedRenderDescriptorSetLayout);
00078     layouts.push_back(raytracingDescriptorSetLayout);
00079     raytracingStage.init(device.get(), layouts);
00080     pathTracing.init(device.get(), layouts);
00081
00082     scene->loadModel(device.get(), graphics_command_pool);
00083     updateTexturesInSharedRenderDescriptorSet();
00084
00085     asManager.createASForScene(device.get(), graphics_command_pool, scene);
00086     create_object_description_buffer();
00087     createRaytracingDescriptorSets();
00088     updateRaytracingDescriptorSets();
00089
00090     gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
00091                                     postStage.getRenderPass(),
00092                                     graphics_command_pool);
00093
00094 } catch (const std::runtime_error& e) {
00095     printf("ERROR: %s\n", e.what());
00096 }
00097 }
```

References allocator, asManager, create\_command\_buffers(), create\_command\_pool(), create\_object\_description\_buffer(), create\_post\_descriptor\_layout(), create\_surface(), create\_uniform\_buffers(), ASManager::createASForScene(), createDescriptorPoolSharedRenderStages(), createRaytracingDescriptorPool(), createRaytracingDescriptorSetLayouts(), createRaytracingDescriptorSets(), createSharedRenderDescriptorSet(), createSharedRenderDescriptorSetLayouts(), createSynchronization(), device, ENABLE\_VALIDATION\_LAYERS, PostStage::getRenderPass(), VulkanInstance::getVulkanInstance(), graphics\_command\_pool, gui, PathTracing::init(), Raytracing::init(), PostStage::init(), Rasterizer::init(), GUI::initializeVulkanContext(), VulkanSwapChain::initVulkanContext(), instance, Scene::loadModel(), pathTracing, post\_descriptor\_set\_layout, postStage, rasterizer, raytracingDescriptorSetLayout, raytracingStage, scene, debug::setupDebugging(), sharedRenderDescriptorSetLayout, surface, updatePostDescriptorSets(), updateRaytracingDescriptorSets(), updateTexturesInSharedRenderDescriptorSet(), updateUniforms(), vulkanSwapChain, and window.

Here is the call graph for this function:



### 5.43.2.2 ~VulkanRenderer()

```
VulkanRenderer::~VulkanRenderer()
```

Definition at line 1229 of file VulkanRenderer.cpp.  
01229 { }

### 5.43.3 Member Function Documentation

#### 5.43.3.1 checkChangedFrameBufferSize()

```
bool VulkanRenderer::checkChangedFrameBufferSize ( ) [private]
```

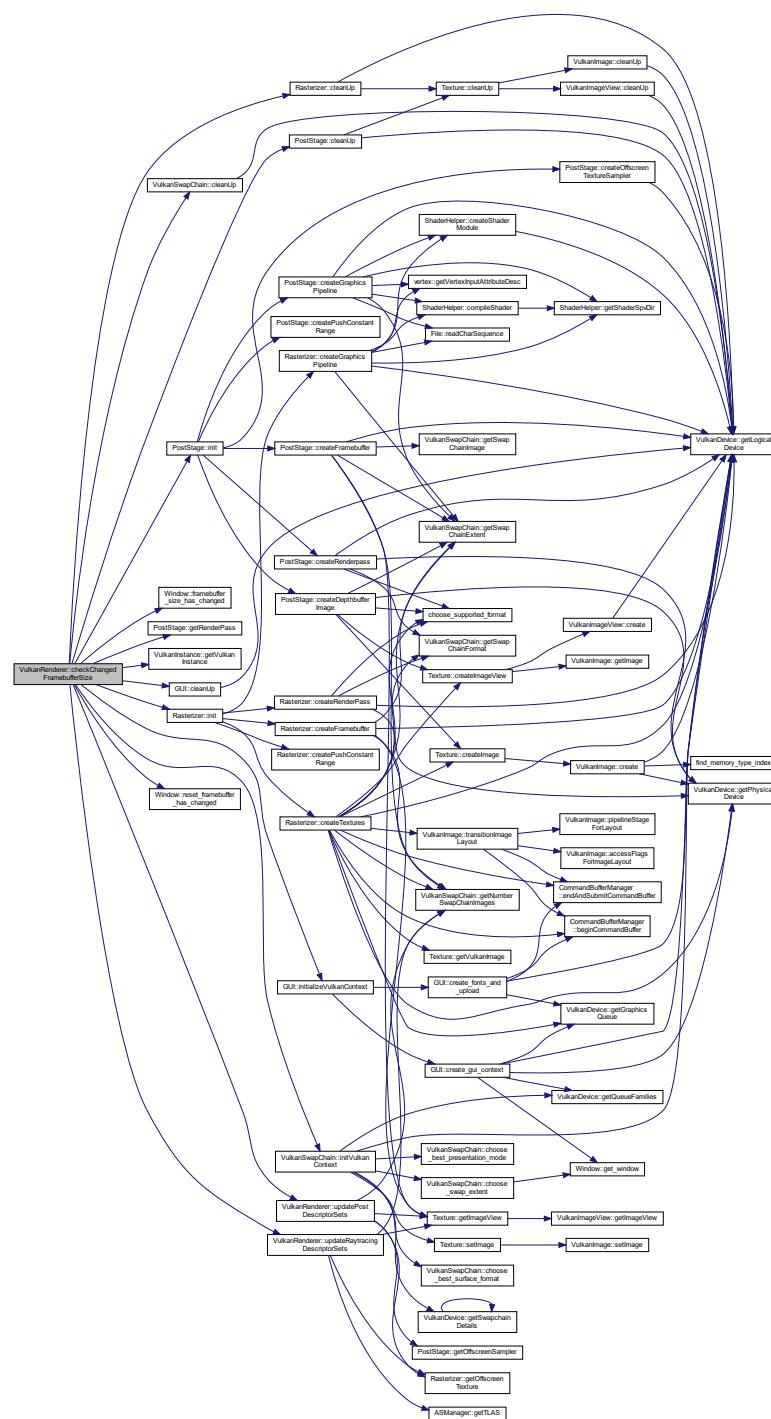
Definition at line 1151 of file [VulkanRenderer.cpp](#).

```
01151     {
01152     if (window->framebuffer_size_has_changed()) {
01153         vkDeviceWaitIdle(device->getLogicalDevice());
01154         vkQueueWaitIdle(device->getGraphicsQueue());
01155
01156         vulkanSwapChain.cleanUp();
01157         vulkanSwapChain.initVulkanContext(device.get(), window, surface);
01158
01159         std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
01160             sharedRenderDescriptorSetLayout};
01161         rasterizer.cleanUp();
01162         rasterizer.init(device.get(), &vulkanSwapChain, descriptor_set_layouts,
01163                         graphics_command_pool);
01164
01165         // all post
01166         std::vector<VkDescriptorSetLayout> descriptorSets = {
01167             post_descriptor_set_layout};
01168         postStage.cleanUp();
01169         postStage.init(device.get(), &vulkanSwapChain, descriptorSets);
01170
01171         gui->cleanUp();
01172         gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
01173                                         postStage.getRenderPass(),
01174                                         graphics_command_pool);
01175
01176         current_frame = 0;
01177
01178         updatePostDescriptorSets();
01179         updateRaytracingDescriptorSets();
01180
01181         window->reset_framebuffer_has_changed();
01182
01183         return true;
01184     }
01185
01186     return false;
01187 }
```

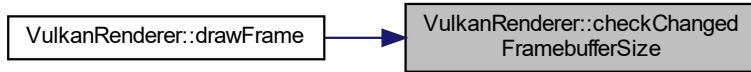
References [GUI::cleanUp\(\)](#), [PostStage::cleanUp\(\)](#), [Rasterizer::cleanUp\(\)](#), [VulkanSwapChain::cleanUp\(\)](#), [current\\_frame](#), [device](#), [Window::framebuffer\\_size\\_has\\_changed\(\)](#), [PostStage::getRenderPass\(\)](#), [VulkanInstance::getVulkanInstance\(\)](#), [graphics\\_command\\_pool](#), [gui](#), [PostStage::init\(\)](#), [Rasterizer::init\(\)](#), [GUI::initializeVulkanContext\(\)](#), [VulkanSwapChain::initVulkanContext\(\)](#), [instance](#), [post\\_descriptor\\_set\\_layout](#), [postStage](#), [rasterizer](#), [Window::reset\\_framebuffer\\_has\\_changed\(\)](#), [sharedRenderDescriptorSetLayout](#), [surface](#), [updatePostDescriptorSets\(\)](#), [updateRaytracingDescriptorSets\(\)](#), [vulkanSwapChain](#), and [window](#).

Referenced by [drawFrame\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.2 cleanUp()

```
void VulkanRenderer::cleanUp( )
```

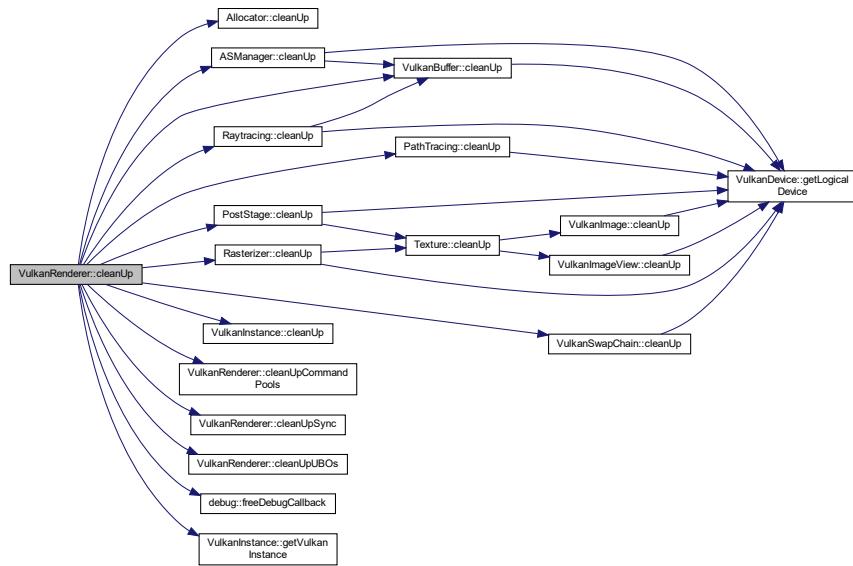
Definition at line 1189 of file [VulkanRenderer.cpp](#).

```

01189     {
01190     cleanUpUBOs();
01191
01192     rasterizer.cleanUp();
01193     raytracingStage.cleanUp();
01194     postStage.cleanUp();
01195     pathTracing.cleanUp();
01196
01197     objectDescriptionBuffer.cleanUp();
01198     asManager.cleanUp();
01199
01200     vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01201         raytracingDescriptorSetLayout, nullptr);
01202     vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01203         post_descriptor_set_layout, nullptr);
01204     vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01205         sharedRenderDescriptorsetLayout, nullptr);
01206     vkDestroyDescriptorPool(device->getLogicalDevice(), post_descriptor_pool,
01207         nullptr);
01208     vkDestroyDescriptorPool(device->getLogicalDevice(),
01209         descriptorPoolSharedRenderStages, nullptr);
01210     vkDestroyDescriptorPool(device->getLogicalDevice(), raytracingDescriptorPool,
01211         nullptr);
01212
01213     vkFreeCommandBuffers(device->getLogicalDevice(), graphics_command_pool,
01214         static_cast<uint32_t>(command_buffers.size()),
01215         command_buffers.data());
01216
01217     cleanUpCommandPools();
01218
01219     cleanUpSync();
01220
01221     vulkanSwapChain.cleanUp();
01222     vkDestroySurfaceKHR(instance.getVulkanInstance(), surface, nullptr);
01223     allocator.cleanUp();
01224     device->cleanUp();
01225     debug::freeDebugCallback(instance.getVulkanInstance());
01226     instance.cleanUp();
01227 }
```

References [allocator](#), [asManager](#), [Allocator::cleanUp\(\)](#), [ASManager::cleanUp\(\)](#), [PathTracing::cleanUp\(\)](#), [PostStage::cleanUp\(\)](#), [Rasterizer::cleanUp\(\)](#), [Raytracing::cleanUp\(\)](#), [VulkanBuffer::cleanUp\(\)](#), [VulkanInstance::cleanUp\(\)](#), [VulkanSwapChain::cleanUp\(\)](#), [cleanUpCommandPools\(\)](#), [cleanUpSync\(\)](#), [cleanUpUBOs\(\)](#), [command\\_buffers](#), [descriptorPoolSharedRenderStages](#), [device](#), [debug::freeDebugCallback\(\)](#), [VulkanInstance::getVulkanInstance\(\)](#), [graphics\\_command\\_pool](#), [instance](#), [objectDescriptionBuffer](#), [pathTracing](#), [post\\_descriptor\\_pool](#), [post\\_descriptor\\_set\\_layout](#), [postStage](#), [rasterizer](#), [raytracingDescriptorPool](#), [raytracingDescriptorsetLayout](#), [raytracingStage](#), [sharedRenderDescriptorsetLayout](#), [surface](#), and [vulkanSwapChain](#).

Here is the call graph for this function:



### 5.43.3.3 cleanUpCommandPools()

```
void VulkanRenderer::cleanUpCommandPools() [private]
```

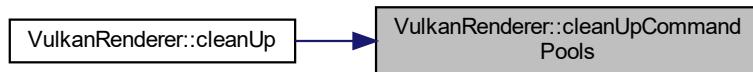
Definition at line 705 of file [VulkanRenderer.cpp](#).

```
00705     {
00706     vkDestroyCommandPool(device->getLogicalDevice(), graphics_command_pool,
00707                           nullptr);
00708     vkDestroyCommandPool(device->getLogicalDevice(), compute_command_pool,
00709                           nullptr);
00710 }
```

References [compute\\_command\\_pool](#), [device](#), and [graphics\\_command\\_pool](#).

Referenced by [cleanUp\(\)](#).

Here is the caller graph for this function:



#### 5.43.3.4 cleanUpSync()

```
void VulkanRenderer::cleanUpSync ( ) [private]
```

Definition at line 420 of file [VulkanRenderer.cpp](#).

```
00420     {
00421     for (int i = 0; i < MAX_FRAME_DRAW; i++) {
00422         vkDestroySemaphore(device->getLogicalDevice(), render_finished[i], nullptr);
00423         vkDestroySemaphore(device->getLogicalDevice(), image_available[i], nullptr);
00424         vkDestroyFence(device->getLogicalDevice(), in_flight_fences[i], nullptr);
00425     }
00426 }
```

References `device`, `image_available`, `in_flight_fences`, `MAX_FRAME_DRAW`, and `render_finished`.

Referenced by [cleanUp\(\)](#).

Here is the caller graph for this function:



#### 5.43.3.5 cleanUpUBOs()

```
void VulkanRenderer::cleanUpUBOs ( ) [private]
```

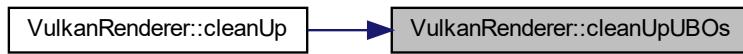
Definition at line 978 of file [VulkanRenderer.cpp](#).

```
00978     {
00979     for (VulkanBuffer vulkanBuffer : globalUBOBuffer) {
00980         vulkanBuffer.cleanUp();
00981     }
00982     for (VulkanBuffer vulkanBuffer : sceneUBOBuffer) {
00983         vulkanBuffer.cleanUp();
00984     }
00985 }
00986 }
```

References `globalUBOBuffer`, and `sceneUBOBuffer`.

Referenced by [cleanUp\(\)](#).

Here is the caller graph for this function:



### 5.43.3.6 `create_command_buffers()`

```
void VulkanRenderer::create_command_buffers ( ) [private]
```

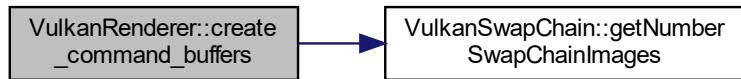
Definition at line 712 of file [VulkanRenderer.cpp](#).

```
00712     {
00713         // resize command buffer count to have one for each framebuffer
00714         command_buffers.resize(vulkanSwapChain.getNumberOfSwapChainImages());
00715
00716         VkCommandBufferAllocateInfo command_buffer_alloc_info{};
00717         command_buffer_alloc_info.sType =
00718             VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
00719         command_buffer_alloc_info.commandPool = graphics_command_pool;
00720         command_buffer_alloc_info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
00721
00722         command_buffer_alloc_info.commandBufferCount =
00723             static_cast<uint32_t>(command_buffers.size());
00724
00725         VkResult result = vkAllocateCommandBuffers(device->getLogicalDevice(),
00726                                         &command_buffer_alloc_info,
00727                                         &command_buffers.data());
00728         ASSERT_VULKAN(result, "Failed to allocate command buffers!")
00729     }
```

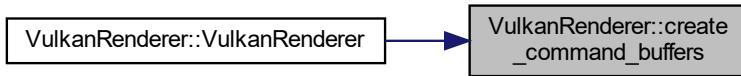
References `command_buffers`, `device`, `VulkanSwapChain::getNumberOfSwapChainImages()`, `graphics_command_pool`, and `vulkanSwapChain`.

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.7 `create_command_pool()`

```
void VulkanRenderer::create_command_pool ( ) [private]
```

Definition at line 664 of file [VulkanRenderer.cpp](#).

```
00664     {
00665         // get indices of queue families from device
00666         QueueFamilyIndices queue_family_indices = device->getQueueFamilies();
00667
00668     {
00669         VkCommandPoolCreateInfo pool_info{};
00670         pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
00671         pool_info.flags =
00672             VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT; // we are ready now to
00673                                         // re-record our
00674                                         // command buffers
00675         pool_info.queueFamilyIndex =
00676             queue_family_indices
00677                 .graphics_family; // queue family type that buffers from this
00678                                         // command pool will use
00679
00680         // create a graphics queue family command pool
00681         VkResult result =
00682             vkCreateCommandPool(device->getLogicalDevice(), &pool_info, nullptr,
00683                 &graphics_command_pool);
00684         ASSERT_VULKAN(result, "Failed to create command pool!")
00685     }
00686
00687     {
00688         VkCommandPoolCreateInfo pool_info{};
00689         pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
00690         pool_info.flags =
00691             VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT; // we are ready now to
00692                                         // re-record our
00693                                         // command buffers
00694         pool_info.queueFamilyIndex =
00695             queue_family_indices.compute_family; // queue family type that buffers
00696                                         // from this command pool will use
00697
00698         // create a graphics queue family command pool
00699         VkResult result = vkCreateCommandPool(
00700             device->getLogicalDevice(), &pool_info, nullptr, &compute_command_pool);
00701         ASSERT_VULKAN(result, "Failed to create command pool!")
00702     }
00703 }
```

References `compute_command_pool`, `QueueFamilyIndices::compute_family`, `device`, `graphics_command_pool`, and `QueueFamilyIndices::graphics_family`.

Referenced by [VulkanRenderer\(\)](#).

Here is the caller graph for this function:



### 5.43.3.8 `create_object_description_buffer()`

```
void VulkanRenderer::create_object_description_buffer ( ) [private]
```

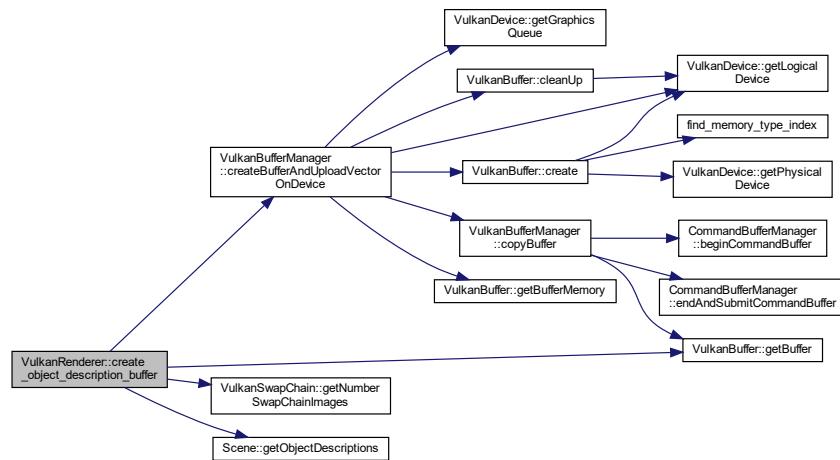
Definition at line 428 of file [VulkanRenderer.cpp](#).

```
00428     std::vector<ObjectDescription> objectDescriptions =
00429         scene->getObjectDescriptions();
00430
00431     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00432         device.get(), graphics_command_pool, objectDescriptionBuffer,
00433         VK_BUFFER_USAGE_TRANSFER_DST_BIT |
00434             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00435             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00436         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00437             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00438         objectDescriptions);
00439
00440
00441     // update the object description set
00442     // update all of descriptor set buffer bindings
00443     for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00444         VkDescriptorBufferInfo object_descriptions_buffer_info{};
00445         // image_info.sampler = VK_DESCRIPTOR_TYPE_SAMPLER;
00446         object_descriptions_buffer_info.buffer =
00447             objectDescriptionBuffer.getBuffer();
00448         object_descriptions_buffer_info.offset = 0;
00449         object_descriptions_buffer_info.range = VK_WHOLE_SIZE;
00450
00451         VkWriteDescriptorSet descriptor_object_descriptions_writer{};
00452         descriptor_object_descriptions_writer.sType =
00453             VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00454         descriptor_object_descriptions_writer.pNext = nullptr;
00455         descriptor_object_descriptions_writer.dstSet = sharedRenderDescriptorSet[i];
00456         descriptor_object_descriptions_writer.dstBinding =
00457             OBJECT_DESCRIPTION_BINDING;
00458         descriptor_object_descriptions_writer.dstArrayElement = 0;
00459         descriptor_object_descriptions_writer.descriptorCount = 1;
00460         descriptor_object_descriptions_writer.descriptorType =
00461             VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00462         descriptor_object_descriptions_writer.pImageInfo = nullptr;
00463         descriptor_object_descriptions_writer.pBufferInfo =
00464             &object_descriptions_buffer_info;
00465         descriptor_object_descriptions_writer.pTexelBufferView =
00466             nullptr; // information about buffer data to bind
00467
00468         std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00469             descriptor_object_descriptions_writer};
00470
00471     // update the descriptor sets with new buffer/binding info
00472     vkUpdateDescriptorSets(device->getLogicalDevice(),
00473         static_cast<uint32_t>(write_descriptor_sets.size()),
00474         write_descriptor_sets.data(), 0, nullptr);
00475 }
00476 }
```

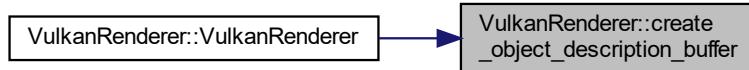
References [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [device](#), [VulkanBuffer::getBuffer\(\)](#), [VulkanSwapChain::getNumberSwapChainImages\(\)](#), [Scene::getObjectDescriptions\(\)](#), [graphics\\_command\\_pool](#), [objectDescriptionBuffer](#), [scene](#), [sharedRenderDescriptorSet](#), [vulkanBufferManager](#), and [vulkanSwapChain](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.9 create\_post\_descriptor\_layout()

```
void VulkanRenderer::create_post_descriptor_layout( ) [private]
```

Definition at line 288 of file [VulkanRenderer.cpp](#).

```

00288 // UNIFORM VALUES DESCRIPTOR SET LAYOUT
00289 // globalUBO Binding info
00290 VkDescriptorSetLayoutBinding post_sampler_layout_binding{};
00291 post_sampler_layout_binding.binding =
00292     0; // binding point in shader (designated by binding number in shader)
00293 post_sampler_layout_binding.descriptorType =
00294     VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER; // type of descriptor
00295                                         // (uniform, dynamic uniform,
00296                                         // image sampler, etc)
00297 post_sampler_layout_binding.descriptorCount =
00298     1; // number of descriptors for binding
00299 post_sampler_layout_binding.stageFlags =
00300     VK_SHADER_STAGE_FRAGMENT_BIT; // we need to say at which shader we bind
00301                                         // this uniform to
00302 post_sampler_layout_binding.pImmutableSamplers =
00303     nullptr; // for texture: can make sampler data unchangeable (immutable)
00304                                         // by specifying in layout
00305
00306 std::vector<VkDescriptorSetLayoutBinding> layout_bindings = {
00307     post_sampler_layout_binding};
00308
00309
  
```

```

00310 // create descriptor set layout with given bindings
00311 VkDescriptorSetLayoutCreateInfo layout_create_info{};
00312 layout_create_info.sType =
00313     VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00314 layout_create_info.bindingCount = static_cast<uint32_t>(
00315     layout_bindings.size()); // only have 1 for the globalUBO
00316 layout_create_info.pBindings =
00317     layout_bindings.data(); // array of binding infos
00318
00319 // create descriptor set layout
00320 VkResult result = vkCreateDescriptorSetLayout(device->getLogicalDevice(),
00321                                             &layout_create_info, nullptr,
00322                                             &post_descriptor_set_layout);
00323 ASSERT_VULKAN(result, "Failed to create descriptor set layout!")
00324
00325 VkDescriptorPoolSize post_pool_size{};
00326 post_pool_size.type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
00327 post_pool_size.descriptorCount = static_cast<uint32_t>(1);
00328
00329 // list of pool sizes
00330 std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {post_pool_size};
00331
00332 VkDescriptorPoolCreateInfo pool_create_info{};
00333 pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00334 pool_create_info.maxSets =
00335     vulkanSwapChain
00336         .getNumberSwapChainImages(); // maximum number of descriptor sets
00337         // that can be created from pool
00338 pool_create_info.poolSizeCount = static_cast<uint32_t>(
00339     descriptor_pool_sizes.size()); // amount of pool sizes being passed
00340 pool_create_info.pPoolSizes =
00341     descriptor_pool_sizes.data(); // pool sizes to create pool with
00342
00343 // create descriptor pool
00344 result = vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
00345                                 nullptr, &post_descriptor_pool);
00346 ASSERT_VULKAN(result, "Failed to create a descriptor pool!")
00347
00348 // resize descriptor set list so one for every buffer
00349 post_descriptor_set.resize(vulkanSwapChain.getNumberSwapChainImages());
00350
00351 std::vector<VkDescriptorSetLayout> set_layouts(
00352     vulkanSwapChain.getNumberSwapChainImages(), post_descriptor_set_layout);
00353
00354 // descriptor set allocation info
00355 VkDescriptorSetAllocateInfo set_alloc_info{};
00356 set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00357 set_alloc_info.descriptorPool =
00358     post_descriptor_pool; // pool to allocate descriptor set from
00359 set_alloc_info.descriptorSetCount =
00360     vulkanSwapChain.getNumberSwapChainImages(); // number of sets to allocate
00361 set_alloc_info.pSetLayouts =
00362     set_layouts.data(); // layouts to use to allocate sets (1:1 relationship)
00363
00364 // allocate descriptor sets (multiple)
00365 result = vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
00366                                     post_descriptor_set.data());
00367 ASSERT_VULKAN(result, "Failed to create descriptor sets!")
00368 }

```

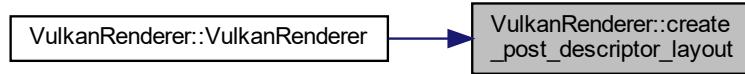
References [device](#), [VulkanSwapChain::getNumberSwapChainImages\(\)](#), [post\\_descriptor\\_pool](#), [post\\_descriptor\\_set](#), [post\\_descriptor\\_set\\_layout](#), and [vulkanSwapChain](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.43.3.10 create\_surface()

```
void VulkanRenderer::create_surface( ) [private]
```

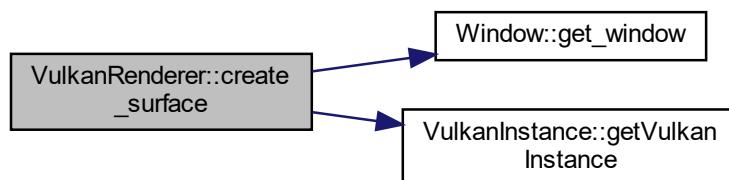
Definition at line 279 of file [VulkanRenderer.cpp](#).

```
00279 {  
00280     // create surface (creates a surface create info struct, runs the create  
00281     // surface function, returns result)  
00282     ASSERT_VULKAN(  
00283         glfwCreateWindowSurface(instance.getVulkanInstance(),  
00284             window->get_window(), nullptr, &surface),  
00285         "Failed to create a surface!");  
00286 }
```

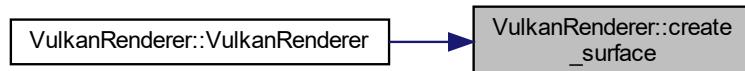
References [Window::get\\_window\(\)](#), [VulkanInstance::getVulkanInstance\(\)](#), [instance](#), [surface](#), and [window](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.11 create\_uniform\_buffers()

```
void VulkanRenderer::create_uniform_buffers ( ) [private]
```

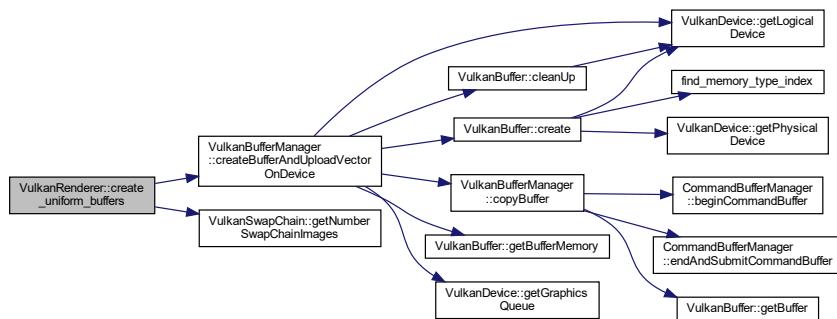
Definition at line 762 of file [VulkanRenderer.cpp](#).

```
00762     {
00763         // one uniform buffer for each image (and by extension, command buffer)
00764         globalUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());
00765         sceneUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());
00766
00767         //// temporary buffer to "stage" vertex data before transferring to GPU
00768         // VulkanBuffer      stagingBuffer;
00769         std::vector<GlobalUBO> globalUBOdata;
00770         globalUBOdata.push_back(globalUBO);
00771
00772         std::vector<SceneUBO> sceneUBOdata;
00773         sceneUBOdata.push_back(sceneUBO);
00774
00775         // create uniform buffers
00776         for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00777             vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00778                 device.get(), graphics_command_pool, globalUBOBuffer[i],
00779                 VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00780                 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, globalUBOdata);
00781
00782             vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00783                 device.get(), graphics_command_pool, sceneUBOBuffer[i],
00784                 VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00785                 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, sceneUBOdata);
00786         }
00787     }
```

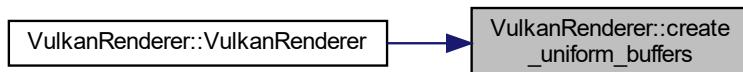
References [VulkanBufferManager::createBufferAndUploadVectorOnDevice\(\)](#), [device](#), [VulkanSwapChain::getNumberSwapChainImages](#), [globalUBO](#), [globalUBOBuffer](#), [graphics\\_command\\_pool](#), [sceneUBO](#), [sceneUBOBuffer](#), [vulkanBufferManager](#), and [vulkanSwapChain](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.12 createDescriptorPoolSharedRenderStages()

```
void VulkanRenderer::createDescriptorPoolSharedRenderStages ( ) [private]
```

Definition at line 789 of file [VulkanRenderer.cpp](#).

```
00789
00790     // CREATE UNIFORM DESCRIPTOR POOL
00791     // type of descriptors + how many descriptors, not descriptor sets (combined
00792     // makes the pool size) ViewProjection Pool
00793     VkDescriptorPoolSize vp_pool_size{};
00794     vp_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00795     vp_pool_size.descriptorCount = static_cast<uint32_t>(globalUBOBuffer.size());
00796
00797     // DIRECTION POOL
00798     VkDescriptorPoolSize directions_pool_size{};
00799     directions_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00800     directions_pool_size.descriptorCount =
00801         static_cast<uint32_t>(sceneUBOBuffer.size());
00802
00803     VkDescriptorPoolSize object_descriptions_pool_size{};
00804     object_descriptions_pool_size.type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00805     object_descriptions_pool_size.descriptorCount =
00806         static_cast<uint32_t>(sizeof(ObjectDescription) * MAX_OBJECTS);
00807
00808     // TEXTURE SAMPLER POOL
00809     VkDescriptorPoolSize sampler_pool_size{};
00810     sampler_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLER;
00811     sampler_pool_size.descriptorCount = MAX_TEXTURE_COUNT;
00812
00813     VkDescriptorPoolSize sampled_image_pool_size{};
00814     sampled_image_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00815     sampled_image_pool_size.descriptorCount = MAX_TEXTURE_COUNT;
00816
00817     // list of pool sizes
00818     std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {
00819         vp_pool_size, directions_pool_size, object_descriptions_pool_size,
00820         sampler_pool_size, sampled_image_pool_size};
00821
00822     VkDescriptorPoolCreateInfo pool_create_info{};
00823     pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00824     pool_create_info.maxSets =
00825         vulkanSwapChain
00826             .getNumberSwapChainImages(); // maximum number of descriptor sets
00827             // that can be created from pool
00828     pool_create_info.poolSizeCount = static_cast<uint32_t>(
00829         descriptor_pool_sizes.size()); // amount of pool sizes being passed
00830     pool_create_info.pPoolSizes =
00831         descriptor_pool_sizes.data(); // pool sizes to create pool with
00832
00833     // create descriptor pool
00834     VkResult result =
00835         vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
00836             nullptr, &descriptorPoolSharedRenderStages);
00837     ASSERT_VULKAN(result, "Failed to create a descriptor pool!")
00838 }
```

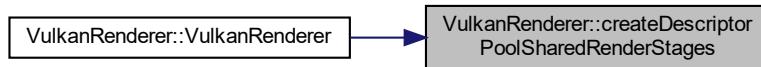
References `descriptorPoolSharedRenderStages`, `device`, `VulkanSwapChain::getNumberSwapChainImages()`, `globalUBOBuffer`, `MAX_OBJECTS`, `MAX_TEXTURE_COUNT`, `sceneUBOBuffer`, and `vulkanSwapChain`.

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.13 createRaytracingDescriptorPool()

void VulkanRenderer::createRaytracingDescriptorPool ( ) [private]

Definition at line 396 of file [VulkanRenderer.cpp](#).

```

00396     {
00397         std::array<VkDescriptorPoolSize, 2> descriptor_pool_sizes{};
00398
00399         descriptor_pool_sizes[0].type = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00400         descriptor_pool_sizes[0].descriptorCount = 1;
00401
00402         descriptor_pool_sizes[1].type = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00403         descriptor_pool_sizes[1].descriptorCount = 1;
00404
00405         VkDescriptorPoolCreateInfo descriptor_pool_create_info{};
00406         descriptor_pool_create_info.sType =
00407             VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00408         descriptor_pool_create_info.poolSizeCount =
00409             static_cast<uint32_t>(descriptor_pool_sizes.size());
00410         descriptor_pool_create_info.pPoolSizes = descriptor_pool_sizes.data();
00411         descriptor_pool_create_info.maxSets =
00412             vulkanSwapChain.getNumberOfSwapChainImages();
00413
00414         VkResult result = vkCreateDescriptorPool(device->getLogicalDevice(),
00415                                             &descriptor_pool_create_info,
00416                                             nullptr, &raytracingDescriptorPool);
00417         ASSERT_VULKAN(result, "Failed to create command pool!")
00418     }
  
```

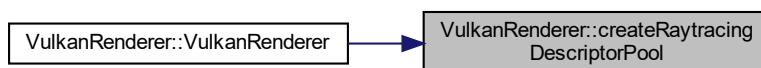
References [device](#), [VulkanSwapChain::getNumberOfSwapChainImages\(\)](#), [raytracingDescriptorPool](#), and [vulkanSwapChain](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.14 createRaytracingDescriptorSetLayouts()

```
void VulkanRenderer::createRaytracingDescriptorSetLayouts ( ) [private]
```

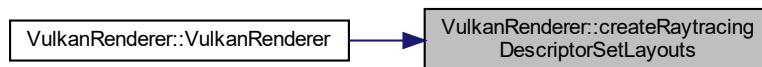
Definition at line 478 of file [VulkanRenderer.cpp](#).

```
00478     {
00479         std::array<VkDescriptorSetLayoutBinding, 2>
00480             descriptor_set_layout_bindings{};
00481
00482         // here comes the top level acceleration structure
00483         descriptor_set_layout_bindings[0].binding = TLAS_BINDING;
00484         descriptor_set_layout_bindings[0].descriptorCount = 1;
00485         descriptor_set_layout_bindings[0].descriptorType =
00486             VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00487         descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;
00488         // load them into the raygeneration and closest hit shader
00489         descriptor_set_layout_bindings[0].stageFlags =
00490             VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00491             VK_SHADER_STAGE_COMPUTE_BIT;
00492         // here comes to previous rendered image
00493         descriptor_set_layout_bindings[1].binding = OUT_IMAGE_BINDING;
00494         descriptor_set_layout_bindings[1].descriptorCount = 1;
00495         descriptor_set_layout_bindings[1].descriptorType =
00496             VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00497         descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;
00498         // load them into the raygeneration and closest hit shader
00499         descriptor_set_layout_bindings[1].stageFlags =
00500             VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00501             VK_SHADER_STAGE_COMPUTE_BIT;
00502
00503         VkDescriptorSetLayoutCreateInfo descriptor_set_layout_create_info{};
00504         descriptor_set_layout_create_info.sType =
00505             VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00506         descriptor_set_layout_create_info.bindingCount =
00507             static_cast<uint32_t>(descriptor_set_layout_bindings.size());
00508         descriptor_set_layout_create_info.pBindings =
00509             descriptor_set_layout_bindings.data();
00510
00511         VkResult result = vkCreateDescriptorSetLayout(
00512             device->getLogicalDevice(), &descriptor_set_layout_create_info, nullptr,
00513             &raytracingDescriptorsetLayout);
00514         ASSERT_VULKAN(result, "Failed to create raytracing descriptor set layout!")
00515     }
00516 }
```

References [device](#), and [raytracingDescriptorsetLayout](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the caller graph for this function:



### 5.43.3.15 `createRaytracingDescriptorSets()`

```
void VulkanRenderer::createRaytracingDescriptorSets () [private]
```

Definition at line 519 of file [VulkanRenderer.cpp](#).

```
00519                                     {
00520     // resize descriptor set list so one for every buffer
00521     raytracingDescriptorSet.resize(vulkanSwapChain.getNumberSwapChainImages());
00522
00523     std::vector<VkDescriptorSetLayout> set_layouts(
00524         vulkanSwapChain.getNumberSwapChainImages(),
00525         raytracingDescriptorsetLayout);
00526
00527     VkDescriptorSetAllocateInfo descriptor_set_allocate_info{};
00528     descriptor_set_allocate_info.sType =
00529         VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00530 ;
00531     descriptor_set_allocate_info.descriptorPool = raytracingDescriptorPool;
00532     descriptor_set_allocate_info.descriptorSetCount =
00533         vulkanSwapChain.getNumberSwapChainImages();
00534     descriptor_set_allocate_info.pSetLayouts = set_layouts.data();
00535
00536     VkResult result = vkAllocateDescriptorSets(device->getLogicalDevice(),
00537                                                 &descriptor_set_allocate_info,
00538                                                 raytracingDescriptorSet.data());
00539     ASSERT_VULKAN(result, "Failed to allocate raytracing descriptor set!")
00540 }
```

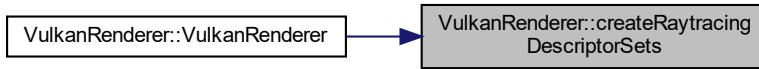
References `device`, `VulkanSwapChain::getNumberSwapChainImages()`, `raytracingDescriptorPool`, `raytracingDescriptorSet`, `raytracingDescriptorsetLayout`, and `vulkanSwapChain`.

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.16 createSharedRenderDescriptorSet()

```
void VulkanRenderer::createSharedRenderDescriptorSet ( ) [private]
```

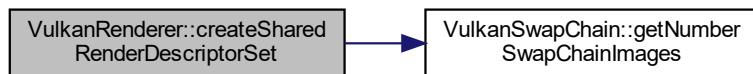
Definition at line 840 of file [VulkanRenderer.cpp](#).

```
00840
00841     // resize descriptor set list so one for every buffer
00842     sharedRenderDescriptorSet.resize(vulkanSwapChain.getNumberSwapChainImages());
00843
00844     std::vector<VkDescriptorSetLayout> set_layouts(
00845         vulkanSwapChain.getNumberSwapChainImages(),
00846         sharedRenderDescriptorSetLayout);
00847
00848     // descriptor set allocation info
00849     VkDescriptorSetAllocateInfo set_alloc_info{};
00850     set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00851     set_alloc_info.descriptorPool =
00852         descriptorPoolSharedRenderStages; // pool to allocate descriptor set from
00853     set_alloc_info.descriptorSetCount =
00854         vulkanSwapChain.getNumberSwapChainImages(); // number of sets to allocate
00855     set_alloc_info.pSetLayouts =
00856         set_layouts.data(); // layouts to use to allocate sets (1:1 relationship)
00857
00858     // allocate descriptor sets (multiple)
00859     VkResult result =
00860         vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
00861             sharedRenderDescriptorSet.data());
00862     ASSERT_VULKAN(result, "Failed to create descriptor sets!")
00863
00864     // update all of descriptor set buffer bindings
00865     for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00866         // VIEW PROJECTION DESCRIPTOR
00867         // buffer info and data offset info
00868         VkDescriptorBufferInfo globalUBO_buffer_info{};
00869         globalUBO_buffer_info.buffer =
00870             globalUBOBuffer[i].getBuffer(); // buffer to get data from
00871         globalUBO_buffer_info.offset = 0; // position of start of data
00872         globalUBO_buffer_info.range = sizeof(globalUBO); // size of data
00873
00874         // data about connection between binding and buffer
00875         VkWriteDescriptorSet globalUBO_set_write{};
00876         globalUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00877         globalUBO_set_write.dstSet =
00878             sharedRenderDescriptorSet[i]; // descriptor set to update
00879         globalUBO_set_write.dstBinding =
00880             0; // binding to update (matches with binding on layout/shader)
00881         globalUBO_set_write.dstArrayElement = 0; // index in array to update
00882         globalUBO_set_write.descriptorType =
00883             VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER; // type of descriptor
00884         globalUBO_set_write.descriptorCount = 1; // amount to update
00885         globalUBO_set_write.pBufferInfo =
00886             &globalUBO_buffer_info; // information about buffer data to bind
00887
00888         // VIEW PROJECTION DESCRIPTOR
00889         // buffer info and data offset info
00890         VkDescriptorBufferInfo sceneUBO_buffer_info{};
00891         sceneUBO_buffer_info.buffer =
00892             sceneUBOBuffer[i].getBuffer(); // buffer to get data from
00893         sceneUBO_buffer_info.offset = 0; // position of start of data
00894         sceneUBO_buffer_info.range = sizeof(sceneUBO); // size of data
00895
00896         // data about connection between binding and buffer
00897         VkWriteDescriptorSet sceneUBO_set_write{};
00898         sceneUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00899         sceneUBO_set_write.dstSet =
00900             sharedRenderDescriptorSet[i]; // descriptor set to update
00901         sceneUBO_set_write.dstBinding =
00902             1; // binding to update (matches with binding on layout/shader)
00903         sceneUBO_set_write.dstArrayElement = 0; // index in array to update
00904         sceneUBO_set_write.descriptorType =
00905             VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER; // type of descriptor
00906         sceneUBO_set_write.descriptorCount = 1; // amount to update
00907         sceneUBO_set_write.pBufferInfo =
00908             &sceneUBO_buffer_info; // information about buffer data to bind
00909
00910         std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00911             globalUBO_set_write, sceneUBO_set_write};
00912
00913         // update the descriptor sets with new buffer/binding info
00914         vkUpdateDescriptorSets(device->getLogicalDevice(),
00915             static_cast<uint32_t>(write_descriptor_sets.size()),
00916             write_descriptor_sets.data(), 0, nullptr);
00917     }
00918 }
```

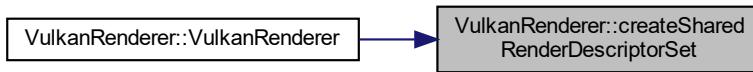
References `descriptorPoolSharedRenderStages`, `device`, `VulkanSwapChain::getNumberSwapChainImages()`, `globalUBO`, `globalUBOBuffer`, `sceneUBO`, `sceneUBOBuffer`, `sharedRenderDescriptorSet`, `sharedRenderDescriptorSetLayout`, and `vulkanSwapChain`.

Referenced by `VulkanRenderer()`.

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.43.3.17 `createSharedRenderDescriptorSetLayouts()`

```
void VulkanRenderer::createSharedRenderDescriptorSetLayouts( ) [private]
```

Definition at line 596 of file `VulkanRenderer.cpp`.

```

00596
00597     std::array<VkDescriptorSetLayoutBinding, 5> descriptor_set_layout_bindings{};
00598     // UNIFORM VALUES DESCRIPTOR SET LAYOUT
00599     // globalUBO Binding info
00600     descriptor_set_layout_bindings[0].binding = globalUBO_BINDING;
00601     descriptor_set_layout_bindings[0].descriptorType =
00602         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00603     descriptor_set_layout_bindings[0].descriptorCount = 1;
00604     descriptor_set_layout_bindings[0].stageFlags =
00605         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00606         VK_SHADER_STAGE_COMPUTE_BIT;
00607     descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;
00608
00609     // our model matrix which updates every frame for each object
00610     descriptor_set_layout_bindings[1].binding = sceneUBO_BINDING;
00611     descriptor_set_layout_bindings[1].descriptorType =
00612         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00613     descriptor_set_layout_bindings[1].descriptorCount = 1;
00614     descriptor_set_layout_bindings[1].stageFlags =
00615         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
00616         VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00617         VK_SHADER_STAGE_COMPUTE_BIT;
00618     descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;
00619
00620     descriptor_set_layout_bindings[2].binding = OBJECT_DESCRIPTION_BINDING;
00621     descriptor_set_layout_bindings[2].descriptorCount = 1;
00622     descriptor_set_layout_bindings[2].descriptorType =
00623         VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;

```

```

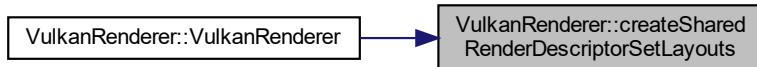
00624     descriptor_set_layout_bindings[2].pImmutableSamplers = nullptr;
00625     // load them into the raygeneration and closest hit shader
00626     descriptor_set_layout_bindings[2].stageFlags =
00627         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
00628         VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR | VK_SHADER_STAGE_COMPUTE_BIT;
00629
00630     // CREATE TEXTURE SAMPLER DESCRIPTOR SET LAYOUT
00631     // texture binding info
00632     descriptor_set_layout_bindings[3].binding = SAMPLER_BINDING;
00633     descriptor_set_layout_bindings[3].descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
00634     descriptor_set_layout_bindings[3].descriptorCount = MAX_TEXTURE_COUNT;
00635     descriptor_set_layout_bindings[3].stageFlags =
00636         VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00637         VK_SHADER_STAGE_COMPUTE_BIT;
00638     descriptor_set_layout_bindings[3].pImmutableSamplers = nullptr;
00639
00640     descriptor_set_layout_bindings[4].binding = TEXTURES_BINDING;
00641     descriptor_set_layout_bindings[4].descriptorType =
00642         VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00643     descriptor_set_layout_bindings[4].descriptorCount = MAX_TEXTURE_COUNT;
00644     descriptor_set_layout_bindings[4].stageFlags =
00645         VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00646         VK_SHADER_STAGE_COMPUTE_BIT;
00647     descriptor_set_layout_bindings[4].pImmutableSamplers = nullptr;
00648
00649     // create descriptor set layout with given bindings
00650     VkDescriptorSetLayoutCreateInfo layout_create_info{};
00651     layout_create_info.sType =
00652         VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00653     layout_create_info.bindingCount =
00654         static_cast<uint32_t>(descriptor_set_layout_bindings.size());
00655     layout_create_info.pBindings = descriptor_set_layout_bindings.data();
00656
00657     // create descriptor set layout
00658     VkResult result = vkCreateDescriptorSetLayout(
00659         device->getLogicalDevice(), &layout_create_info, nullptr,
00660         &sharedRenderDescriptorSetLayout);
00661     ASSERT_VULKAN(result, "Failed to create descriptor set layout!")
00662 }

```

References [device](#), [MAX\\_TEXTURE\\_COUNT](#), and [sharedRenderDescriptorSetLayout](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the caller graph for this function:



### 5.43.3.18 createSynchronization()

```
void VulkanRenderer::createSynchronization ( ) [private]
```

Definition at line 731 of file [VulkanRenderer.cpp](#).

```

00731     {
00732     image_available.resize(vulkanSwapChain.getNumberSwapChainImages(),
00733                             VK_NULL_HANDLE);
00734     render_finished.resize(vulkanSwapChain.getNumberSwapChainImages(),
00735                            VK_NULL_HANDLE);
00736     in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
00737                             VK_NULL_HANDLE);
00738     images_in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
00739                                    VK_NULL_HANDLE);

```

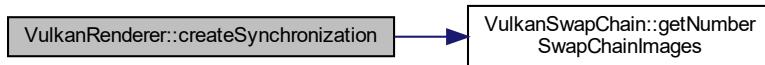
```

00740
00741     // semaphore creation information
00742     VkSemaphoreCreateInfo semaphore_create_info{};
00743     semaphore_create_info.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
00744
00745     // fence creation information
00746     VkFenceCreateInfo fence_create_info{};
00747     fence_create_info.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
00748     fence_create_info.flags = VK_FENCE_CREATE_SIGNALED_BIT;
00749
00750     for (int i = 0; i < MAX_FRAME_DRAWS; i++) {
00751         if ((vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
00752                               nullptr, &image_available[i]) != VK_SUCCESS) ||
00753             (vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
00754                               nullptr, &render_finished[i]) != VK_SUCCESS) ||
00755             (vkCreateFence(device->getLogicalDevice(), &fence_create_info, nullptr,
00756                           &in_flight_fences[i]) != VK_SUCCESS)) {
00757             throw std::runtime_error("Failed to create a semaphore and/or fence!");
00758         }
00759     }
00760 }
```

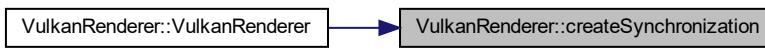
References `device`, `VulkanSwapChain::getNumberSwapChainImages()`, `image_available`, `images_in_flight_fences`, `in_flight_fences`, `MAX_FRAME_DRAWS`, `render_finished`, and `vulkanSwapChain`.

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.19 drawFrame()

```
void VulkanRenderer::drawFrame( )
```

check if previous frame is using this image (i.e. there is its fence to wait on)

Definition at line 152 of file [VulkanRenderer.cpp](#).

```

00152
00153     // We need to skip one frame
00154     // Due to ImGuizmo need to call ImGuizmo::NewFrame() again
00155     // if we recreated swapchain
00156     if (checkChangedFrameBufferSize()) return;
00157 }
```

```

00158 /*1. Get next available image to draw to and set something to signal when
00159     we're finished with the image (a semaphore) wait for given fence to signal
00160     (open) from last draw before continuing*/
00161 VkResult result = vkWaitForFences(device->getLogicalDevice(), 1,
00162                                     &in_flight_fences[current_frame], VK_TRUE,
00163                                     std::numeric_limits<uint64_t>::max());
00164 ASSERT_VULKAN(result, "Failed to wait for fences!")
00165 // -- GET NEXT IMAGE --
00166 uint32_t image_index;
00167 result = vkAcquireNextImageKHR(
00168     device->getLogicalDevice(), vulkanSwapChain.getSwapChain(),
00169     std::numeric_limits<uint64_t>::max(), image_available[current_frame],
00170     VK_NULL_HANDLE, &image_index);
00171
00172 if (result == VK_ERROR_OUT_OF_DATE_KHR) {
00173     // recreate_swap_chain();
00174     return;
00175 }
00176 } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
00177     throw std::runtime_error("Failed to acquire next image!");
00178 }
00179
00180 ///// check if previous frame is using this image (i.e. there is its fence to
00181 ///// wait on)
00182 if (images_in_flight_fences[image_index] != VK_NULL_HANDLE) {
00183     vkWaitForFences(device->getLogicalDevice(), 1,
00184                     &images_in_flight_fences[image_index], VK_TRUE, UINT64_MAX);
00185 }
00186
00187 // mark the image as now being in use by this frame
00188 images_in_flight_fences[image_index] = in_flight_fences[current_frame];
00189
00190 VkCommandBufferBeginInfo buffer_begin_info{};
00191 buffer_begin_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
00192 buffer_begin_info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
00193 // start recording commands to command buffer
00194 result =
00195     vkBeginCommandBuffer(command_buffers[image_index], &buffer_begin_info);
00196 ASSERT_VULKAN(result, "Failed to start recording a command buffer!")
00197
00198 update_uniform_buffers(image_index);
00199
00200 GUIRendererSharedVars& guiRendererSharedVars =
00201     gui->getGuiRendererSharedVars();
00202 if (guiRendererSharedVars.raytracing)
00203     update_raytracing_descriptor_set(image_index);
00204
00205 record_commands(image_index);
00206
00207 // stop recording to command buffer
00208 result = vkEndCommandBuffer(command_buffers[image_index]);
00209 ASSERT_VULKAN(result, "Failed to stop recording a command buffer!")
00210
00211 // 2. Submit command buffer to queue for execution, making sure it waits for
00212 // the image to be signalled as available before drawing and signals when it
00213 // has finished rendering
00214 // -- SUBMIT COMMAND BUFFER TO RENDER --
00215 VkSubmitInfo submit_info{};
00216 submit_info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
00217 submit_info.waitSemaphoreCount = 1; // number of semaphores to wait on
00218 submit_info.pWaitSemaphores =
00219     &image_available[current_frame]; // list of semaphores to wait on
00220
00221 VkPipelineStageFlags wait_stages =
00222
00223     VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT /*|
00224         VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT |
00225         VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR*/
00226
00227 };
00228
00229 submit_info.pWaitDstStageMask =
00230     &wait_stages; // stages to check semaphores at
00231
00232 submit_info.commandBufferCount = 1; // number of command buffers to submit
00233 submit_info.pCommandBuffers =
00234     &command_buffers[image_index]; // command buffer to submit
00235 submit_info.signalSemaphoreCount = 1; // number of semaphores to signal
00236 submit_info.pSignalSemaphores =
00237     &render_finished[current_frame]; // semaphores to signal when command
00238                                // buffer finishes
00239
00240 result = vkResetFences(device->getLogicalDevice(), 1,
00241                         &in_flight_fences[current_frame]);
00242 ASSERT_VULKAN(result, "Failed to reset fences!")
00243
00244 // submit command buffer to queue

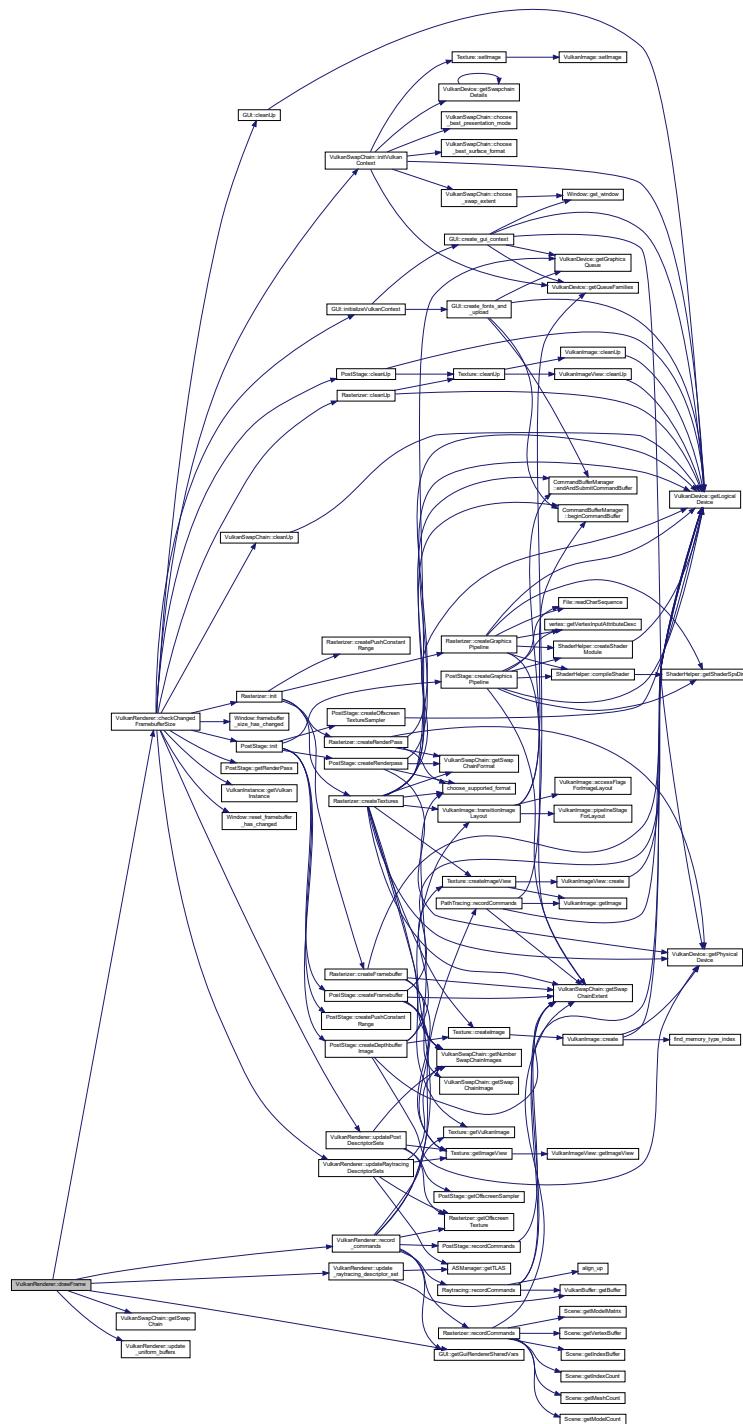
```

```

00245     result = vkQueueSubmit(device->getGraphicsQueue(), 1, &submit_info,
00246                             in_flight_fences[current_frame]);
00247     ASSERT_VULKAN(result, "Failed to submit command buffer to queue!")
00248
00249     // 3. Present image to screen when it has signalled finished rendering
00250     // -- PRESENT RENDERED IMAGE TO SCREEN --
00251     VkPresentInfoKHR present_info{};
00252     present_info.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
00253     present_info.waitSemaphoreCount = 1; // number of semaphores to wait on
00254     present_info.pWaitSemaphores =
00255         &render_finished[current_frame]; // semaphores to wait on
00256     present_info.swapchainCount = 1; // number of swapchains to present to
00257     const VkSwapchainKHR swapchain = vulkanSwapChain.getSwapChain();
00258     present_info.pSwapchains = &swapchain; // swapchains to present images to
00259     present_info.pImageIndices =
00260         &image_index; // index of images in swapchain to present
00261
00262     result = vkQueuePresentKHR(device->getPresentationQueue(), &present_info);
00263
00264     if (result == VK_ERROR_OUT_OF_DATE_KHR) {
00265         // recreate_swap_chain();
00266         return;
00267     }
00268     else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
00269         throw std::runtime_error("Failed to acquire next image!");
00270     }
00271     if (result != VK_SUCCESS) {
00272         throw std::runtime_error("Failed to submit to present queue!");
00273     }
00274
00275     current_frame = (current_frame + 1) % MAX_FRAME_DRAWS;
00276
00277 }
```

References [checkChangedFramebufferSize\(\)](#), [command\\_buffers](#), [current\\_frame](#), [device](#), [GUI::getGuiRendererSharedVars\(\)](#), [VulkanSwapChain::getSwapChain\(\)](#), [gui](#), [image\\_available](#), [images\\_in\\_flight\\_fences](#), [in\\_flight\\_fences](#), [MAX\\_FRAME\\_DRAWS](#), [GUIRendererSharedVars::raytracing](#), [record\\_commands\(\)](#), [render\\_finished](#), [update\\_raytracing\\_descriptor\\_set\(\)](#), [update\\_uniform\\_buffers\(\)](#), and [vulkanSwapChain](#).

Here is the call graph for this function:



#### 5.43.3.20 finishAllRenderCommands()

```
void VulkanRenderer::finishAllRenderCommands ( )
```

Definition at line 130 of file [VulkanRenderer.cpp](#).

```
00130           {  
00131     vkDeviceWaitIdle(device->getLogicalDevice());  
00132 }
```

References [device](#).

### 5.43.3.21 record\_commands()

```
void VulkanRenderer::record_commands (  
    uint32_t image_index ) [private]
```

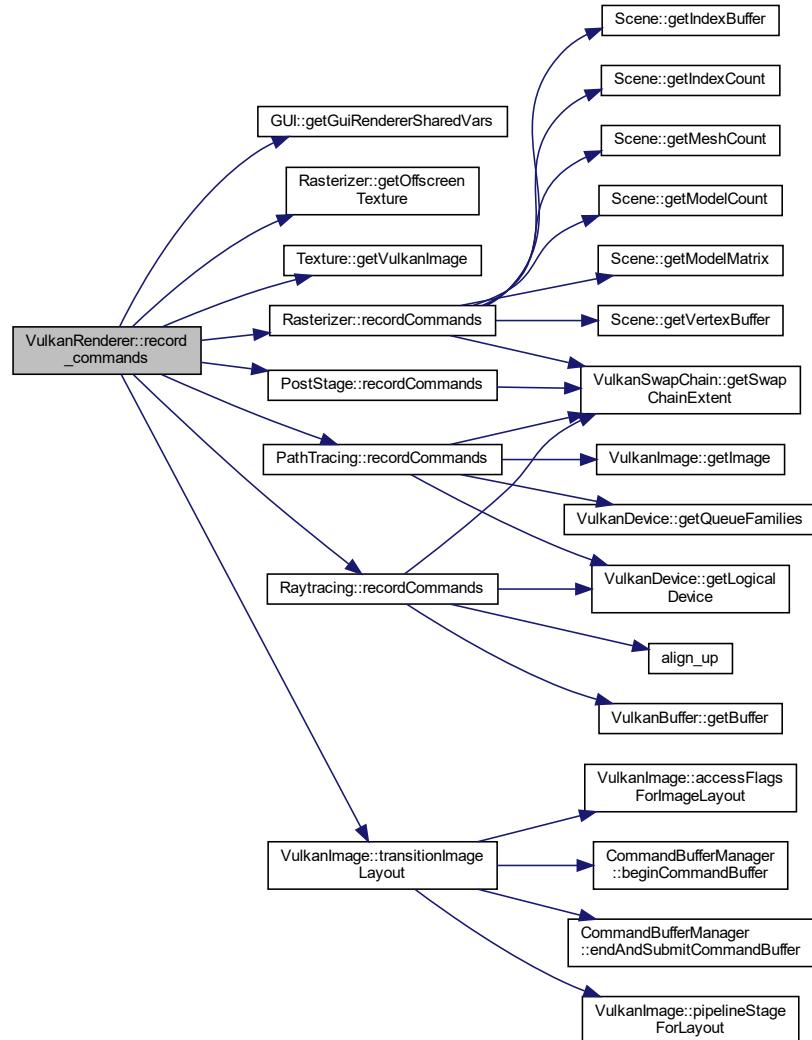
Definition at line 1110 of file [VulkanRenderer.cpp](#).

```
01110           {  
01111     Texture& renderResult = rasterizer.getOffscreenTexture(image_index);  
01112     VulkanImage& vulkanImage = renderResult.getVulkanImage();  
01113  
01114     GUIRendererSharedVars& guiRendererSharedVars =  
01115         gui->getGuiRendererSharedVars();  
01116     if (guiRendererSharedVars.raytracing) {  
01117         std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],  
01118                                         raytracingDescriptorSet[image_index]};  
01119         raytracingStage.recordCommands(command_buffers[image_index],  
01120                                         &vulkanSwapChain, sets);  
01121  
01122     } else if (guiRendererSharedVars.pathTracing) {  
01123         std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],  
01124                                         raytracingDescriptorSet[image_index]};  
01125  
01126         pathTracing.recordCommands(command_buffers[image_index], image_index,  
01127                                         vulkanImage, &vulkanSwapChain, sets);  
01128  
01129     } else {  
01130         std::vector<VkDescriptorSet> descriptorSets = {  
01131             sharedRenderDescriptorSet[image_index]};  
01132  
01133         rasterizer.recordCommands(command_buffers[image_index], image_index, scene,  
01134                                         descriptorSets);  
01135     }  
01136  
01137     vulkanImage.transitionImageLayout (  
01138         command_buffers[image_index], VK_IMAGE_LAYOUT_GENERAL,  
01139         VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);  
01140  
01141     std::vector<VkDescriptorSet> descriptorSets = {  
01142         post_descriptor_set[image_index]};  
01143     postStage.recordCommands(command_buffers[image_index], image_index,  
01144                                         descriptorSets);  
01145  
01146     vulkanImage.transitionImageLayout (  
01147         command_buffers[image_index], VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,  
01148         VK_IMAGE_LAYOUT_GENERAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);  
01149 }
```

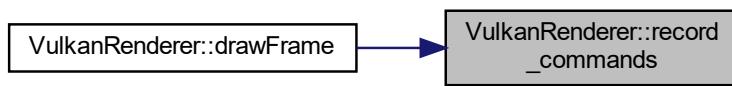
References [command\\_buffers](#), [GUI::getGuiRendererSharedVars\(\)](#), [Rasterizer::getOffscreenTexture\(\)](#), [Texture::getVulkanImage\(\)](#), [gui](#), [GUIRendererSharedVars::pathTracing](#), [pathTracing](#), [post\\_descriptor\\_set](#), [postStage](#), [rasterizer](#), [GUIRendererSharedVars::raytracing](#), [raytracingDescriptorSet](#), [raytracingStage](#), [PostStage::recordCommands\(\)](#), [Rasterizer::recordCommands\(\)](#), [PathTracing::recordCommands\(\)](#), [Raytracing::recordCommands\(\)](#), [scene](#), [sharedRenderDescriptorSet](#), [VulkanImage::transitionImageLayout](#) and [vulkanSwapChain](#).

Referenced by [drawFrame\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.22 shaderHotReload()

```
void VulkanRenderer::shaderHotReload( ) [private]
```

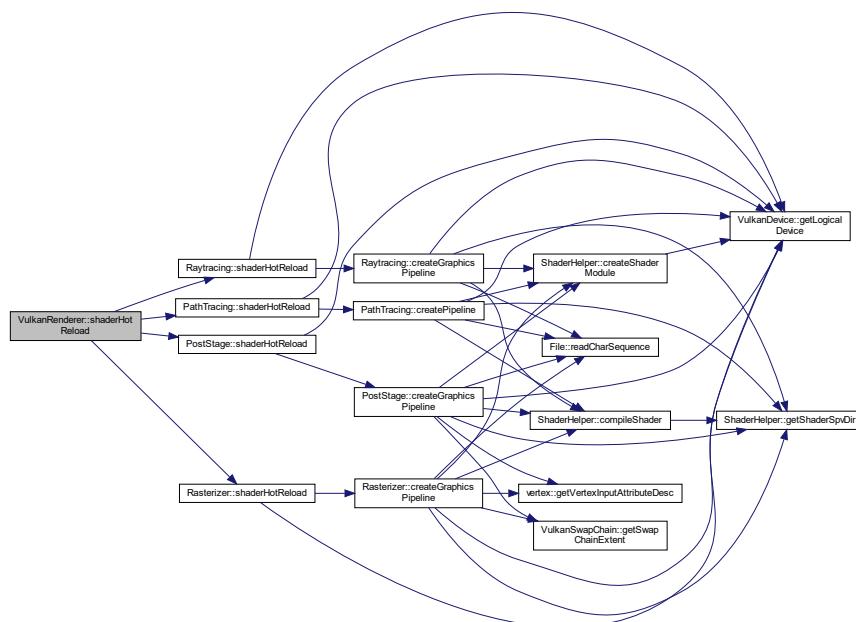
Definition at line 134 of file [VulkanRenderer.cpp](#).

```
00134     {
00135     // wait until no actions being run on device before destroying
00136     vkDeviceWaitIdle(device->getLogicalDevice());
00137
00138     std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
00139         sharedRenderDescriptorsetLayout};
00140     rasterizer.shaderHotReload(descriptor_set_layouts);
00141
00142     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
00143         post_descriptor_set_layout};
00144     postStage.shaderHotReload(descriptor_set_layouts_post);
00145
00146     std::vector<VkDescriptorSetLayout> layouts = {sharedRenderDescriptorsetLayout,
00147                                                 raytracingDescriptorsetLayout};
00148     raytracingStage.shaderHotReload(layouts);
00149     pathTracing.shaderHotReload(layouts);
00150 }
```

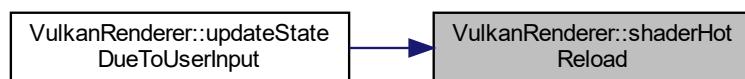
References [device](#), [pathTracing](#), [post\\_descriptor\\_set\\_layout](#), [postStage](#), [rasterizer](#), [raytracingDescriptorsetLayout](#), [raytracingStage](#), [PathTracing::shaderHotReload\(\)](#), [PostStage::shaderHotReload\(\)](#), [Rasterizer::shaderHotReload\(\)](#), [Raytracing::shaderHotReload\(\)](#), and [sharedRenderDescriptorsetLayout](#).

Referenced by [updateStateDueToUserInput\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.23 update\_raytracing\_descriptor\_set()

```
void VulkanRenderer::update_raytracing_descriptor_set (
    uint32_t image_index )
```

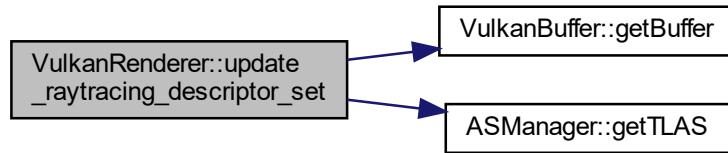
Definition at line 1059 of file [VulkanRenderer.cpp](#).

```
01059     VkWriteDescriptorSetAccelerationStructureKHR
01060     descriptor_set_acceleration_structure{};
01061     descriptor_set_acceleration_structure.sType =
01062         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
01063     descriptor_set_acceleration_structure.pNext = nullptr;
01064     descriptor_set_acceleration_structure.accelerationStructureCount = 1;
01065     VkAccelerationStructureKHR& tlasAS = asManager.getTLAS\(\);
01066     descriptor_set_acceleration_structure.pAccelerationStructures = &tlasAS;
01067
01068     VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
01069     write_descriptor_set_acceleration_structure.sType =
01070         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
01071     write_descriptor_set_acceleration_structure.pNext =
01072         &descriptor_set_acceleration_structure;
01073     write_descriptor_set_acceleration_structure.dstSet =
01074         raytracingDescriptorSet[image_index];
01075     write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
01076     write_descriptor_set_acceleration_structure.dstArrayElement = 0;
01077     write_descriptor_set_acceleration_structure.descriptorCount = 1;
01078     write_descriptor_set_acceleration_structure.descriptorType =
01079         VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
01080     write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
01081     write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
01082     write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;
01083
01084     VkDescriptorBufferInfo object_description_buffer_info{};
01085     object_description_buffer_info.buffer = objectDescriptionBuffer.getBuffer\(\);
01086     object_description_buffer_info.offset = 0;
01087     object_description_buffer_info.range = VK_WHOLE_SIZE;
01088
01089     VkWriteDescriptorSet object_description_buffer_write{};
01090     object_description_buffer_write.sType =
01091         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
01092     object_description_buffer_write.dstSet =
01093         sharedRenderDescriptorSet[image_index];
01094     object_description_buffer_write.descriptorType =
01095         VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
01096     object_description_buffer_write.dstBinding = OBJECT_DESCRIPTION_BINDING;
01097     object_description_buffer_write.pBufferInfo = &object_description_buffer_info;
01098     object_description_buffer_write.descriptorCount = 1;
01099
01100     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
01101         write_descriptor_set_acceleration_structure,
01102         object_description_buffer_write};
01103
01104     vkUpdateDescriptorSets(device->getLogicalDevice(),
01105                             static_cast<uint32_t>(write_descriptor_sets.size()),
01106                             write_descriptor_sets.data(), 0, nullptr);
01107
01108 }
```

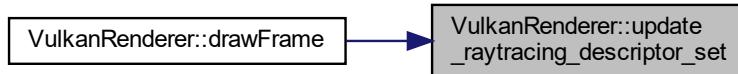
References [asManager](#), [device](#), [VulkanBuffer::getBuffer\(\)](#), [ASManager::getTLAS\(\)](#), [objectDescriptionBuffer](#), [raytracingDescriptorSet](#), and [sharedRenderDescriptorSet](#).

Referenced by [drawFrame\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.43.3.24 update\_uniform\_buffers()

```
void VulkanRenderer::update_uniform_buffers (
    uint32_t image_index ) [private]
```

Definition at line 988 of file [VulkanRenderer.cpp](#).

```

00988     {
00989     auto usage_stage_flags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
00990                           VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR |
00991                           VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
00992
00993     VkBufferMemoryBarrier before_barrier_upv{};
00994     before_barrier_upv.pNext = nullptr;
00995     before_barrier_upv.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
00996     before_barrier_upv.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
00997     before_barrier_upv.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00998     before_barrier_upv.buffer = globalUBOBuffer[image_index].getBuffer();
00999     before_barrier_upv.offset = 0;
01000     before_barrier_upv.size = sizeof(globalUBO);
01001     before_barrier_upv.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01002     before_barrier_upv.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01003
01004     VkBufferMemoryBarrier before_barrier_directions{};
01005     before_barrier_directions.pNext = nullptr;
01006     before_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01007     before_barrier_directions.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
01008     before_barrier_directions.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01009     before_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
01010     before_barrier_directions.offset = 0;
01011     before_barrier_directions.size = sizeof(sceneUBO);
01012     before_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01013     before_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01014
01015     vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
01016                           VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
```

```

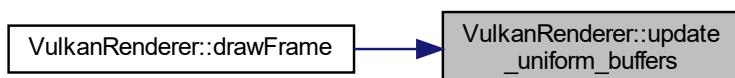
01017             &before_barrier_uvp, 0, nullptr);
01018     vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
01019                           VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
01020                           &before_barrier_directions, 0, nullptr);
01021
01022     vkCmdUpdateBuffer(command_buffers[image_index],
01023                         globalUBOBuffer[image_index].getBuffer(), 0,
01024                         sizeof(GlobalUBO), &globalUBO);
01025     vkCmdUpdateBuffer(command_buffers[image_index],
01026                         sceneUBOBuffer[image_index].getBuffer(), 0,
01027                         sizeof(SceneUBO), &sceneUBO);
01028
01029     VkBufferMemoryBarrier after_barrier_uvp{};
01030     after_barrier_uvp.pNext = nullptr;
01031     after_barrier_uvp.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01032     after_barrier_uvp.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01033     after_barrier_uvp.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
01034     after_barrier_uvp.buffer = globalUBOBuffer[image_index].getBuffer();
01035     after_barrier_uvp.offset = 0;
01036     after_barrier_uvp.size = sizeof(GlobalUBO);
01037     after_barrier_uvp.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01038     after_barrier_uvp.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01039
01040     VkBufferMemoryBarrier after_barrier_directions{};
01041     after_barrier_directions.pNext = nullptr;
01042     after_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01043     after_barrier_directions.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01044     after_barrier_directions.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
01045     after_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
01046     after_barrier_directions.offset = 0;
01047     after_barrier_directions.size = sizeof(SceneUBO);
01048     after_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01049     after_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01050
01051     vkCmdPipelineBarrier(command_buffers[image_index],
01052                           VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
01053                           nullptr, 1, &after_barrier_uvp, 0, nullptr);
01054     vkCmdPipelineBarrier(command_buffers[image_index],
01055                           VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
01056                           nullptr, 1, &after_barrier_directions, 0, nullptr);
01057 }

```

References [command\\_buffers](#), [globalUBO](#), [globalUBOBuffer](#), [sceneUBO](#), and [sceneUBOBuffer](#).

Referenced by [drawFrame\(\)](#).

Here is the caller graph for this function:



#### 5.43.3.25 updatePostDescriptorSets()

```
void VulkanRenderer::updatePostDescriptorSets() [private]
```

Definition at line 370 of file [VulkanRenderer.cpp](#).

```

00370 {
00371     // update all of descriptor set buffer bindings
00372     for (size_t i = 0; i < vulkanSwapChain.getNumberOfSwapChainImages(); i++) {
00373         // texture image info
00374         VkDescriptorImageInfo image_info{};

```

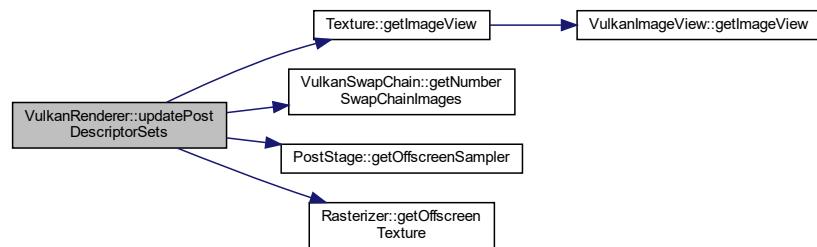
```

00375     image_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00376     Texture& renderResult = rasterizer.getOffscreenTexture(i);
00377     image_info.imageView = renderResult.getImageView();
00378     image_info.sampler = postStage.getOffscreenSampler();
00379
00380     // descriptor write info
00381     VkWriteDescriptorSet descriptor_write{};
00382     descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00383     descriptor_write.dstSet = post_descriptor_set[i];
00384     descriptor_write.dstBinding = 0;
00385     descriptor_write.dstArrayElement = 0;
00386     descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
00387     descriptor_write.descriptorCount = 1;
00388     descriptor_write.pImageInfo = &image_info;
00389
00390     // update new descriptor set
00391     vkUpdateDescriptorSets(device->getLogicalDevice(), 1, &descriptor_write, 0,
00392                             nullptr);
00393 }
00394 }
```

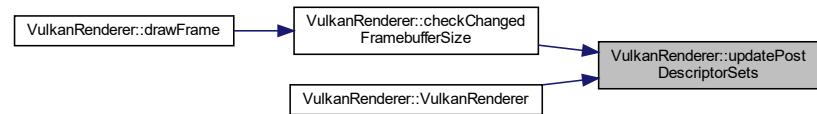
References [device](#), [Texture::getImageView\(\)](#), [VulkanSwapChain::getNumberSwapChainImages\(\)](#), [PostStage::getOffscreenSampler\(\)](#), [Rasterizer::getOffscreenTexture\(\)](#), [post\\_descriptor\\_set](#), [postStage](#), [rasterizer](#), and [vulkanSwapChain](#).

Referenced by [checkChangedFrameBufferSize\(\)](#), and [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.26 updateRaytracingDescriptorSets()

```
void VulkanRenderer::updateRaytracingDescriptorSets( ) [private]
```

Definition at line 542 of file [VulkanRenderer.cpp](#).

```
00542
00543     for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
```

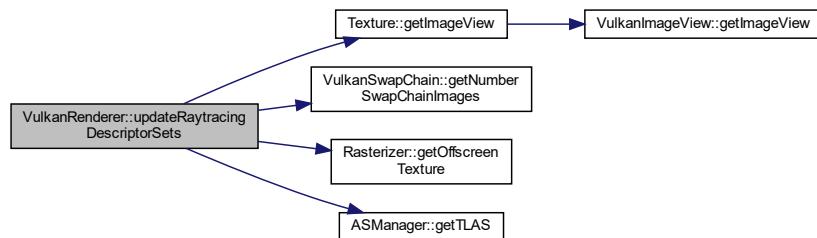
```

00544     VkWriteDescriptorSetAccelerationStructureKHR
00545         descriptor_set_acceleration_structure{};
00546         descriptor_set_acceleration_structure.sType =
00547             VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
00548         descriptor_set_acceleration_structure.pNext = nullptr;
00549         descriptor_set_acceleration_structure.accelerationStructureCount = 1;
00550         VkAccelerationStructureKHR& vulkanTLAS = asManager.getTLAS();
00551         descriptor_set_acceleration_structure.pAccelerationStructures = &vulkanTLAS;
00552
00553     VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
00554     write_descriptor_set_acceleration_structure.sType =
00555         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00556     write_descriptor_set_acceleration_structure.pNext =
00557         &descriptor_set_acceleration_structure;
00558     write_descriptor_set_acceleration_structure.dstSet =
00559         raytracingDescriptorSet[i];
00560     write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
00561     write_descriptor_set_acceleration_structure.dstArrayElement = 0;
00562     write_descriptor_set_acceleration_structure.descriptorCount = 1;
00563     write_descriptor_set_acceleration_structure.descriptorType =
00564         VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00565     write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
00566     write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
00567     write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;
00568
00569     VkDescriptorImageInfo image_info{};
00570     Texture& renderResult = rasterizer.getOffscreenTexture(i);
00571     image_info.imageView = renderResult.getImageView();
00572     image_info.imageLayout = VK_IMAGE_LAYOUT_GENERAL;
00573
00574     VkWriteDescriptorSet descriptor_image_writer{};
00575     descriptor_image_writer.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00576     descriptor_image_writer.pNext = nullptr;
00577     descriptor_image_writer.dstSet = raytracingDescriptorSet[i];
00578     descriptor_image_writer.dstBinding = OUT_IMAGE_BINDING;
00579     descriptor_image_writer.dstArrayElement = 0;
00580     descriptor_image_writer.descriptorCount = 1;
00581     descriptor_image_writer.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00582     descriptor_image_writer.pImageInfo = &image_info;
00583     descriptor_image_writer.pBufferInfo = nullptr;
00584     descriptor_image_writer.pTexelBufferView = nullptr;
00585
00586     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00587         write_descriptor_set_acceleration_structure, descriptor_image_writer};
00588
00589 // update the descriptor sets with new buffer/binding info
00590     vkUpdateDescriptorSets(device->getLogicalDevice(),
00591                         static_cast<uint32_t>(write_descriptor_sets.size()),
00592                         write_descriptor_sets.data(), 0, nullptr);
00593 }
00594 }
```

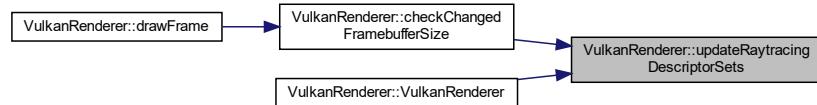
References [asManager](#), [device](#), [Texture::getImageView\(\)](#), [VulkanSwapChain::getNumberSwapChainImages\(\)](#), [Rasterizer::getOffscreenTexture\(\)](#), [ASManager::getTLAS\(\)](#), [rasterizer](#), [raytracingDescriptorSet](#), and [vulkanSwapChain](#).

Referenced by [checkChangedFrameBufferSize\(\)](#), and [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.27 updateStateDueToUserInput()

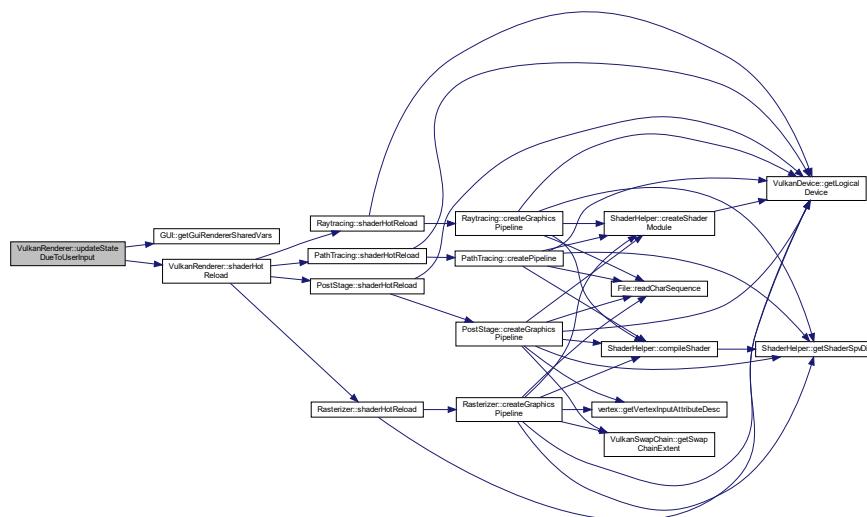
```
void VulkanRenderer::updateStateDueToUserInput (
    GUI * gui )
```

Definition at line 120 of file [VulkanRenderer.cpp](#).

```
00120
00121     GUIRendererSharedVars& guiRendererSharedVars =
00122         gui->getGuiRendererSharedVars();
00123
00124     if (guiRendererSharedVars.shader_hot_reload_triggered) {
00125         shaderHotReload();
00126         guiRendererSharedVars.shader_hot_reload_triggered = false;
00127     }
00128 }
```

References [GUI::getGuiRendererSharedVars\(\)](#), [gui](#), [GUIRendererSharedVars::shader\\_hot\\_reload\\_triggered](#), and [shaderHotReload\(\)](#).

Here is the call graph for this function:



### 5.43.3.28 updateTexturesInSharedRenderDescriptorSet()

```
void VulkanRenderer::updateTexturesInSharedRenderDescriptorSet() [private]
```

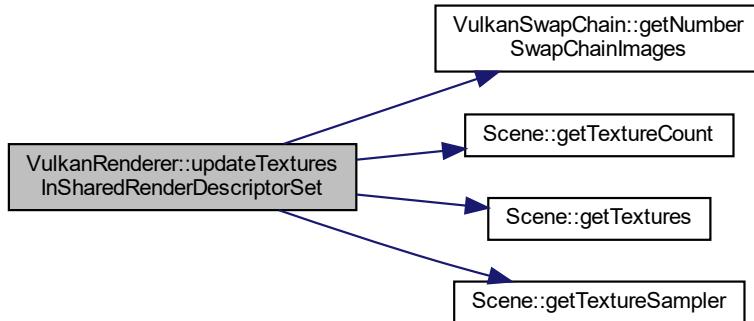
Definition at line 920 of file [VulkanRenderer.cpp](#).

```
00920
00921     std::vector<Texture>& modelTextures = scene->getTextures(0);
00922     std::vector<VkDescriptorImageInfo> image_info_textures;
00923     image_info_textures.resize(scene->getTextureCount(0));
00924     for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
00925         image_info_textures[i].imageLayout =
00926             VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00927         image_info_textures[i].imageView = modelTextures[i].getImageView();
00928         image_info_textures[i].sampler = nullptr;
00929     }
00930
00931     std::vector<VkSampler>& modelTextureSampler = scene->getTextureSampler(0);
00932     std::vector<VkDescriptorImageInfo> image_info_texture_sampler;
00933     image_info_texture_sampler.resize(scene->getTextureCount(0));
00934     for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
00935         image_info_texture_sampler[i].imageLayout =
00936             VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00937         image_info_texture_sampler[i].imageView = nullptr;
00938         image_info_texture_sampler[i].sampler = modelTextureSampler[i];
00939     }
00940
00941     for (uint32_t i = 0; i < vulkanSwapChain.getNumberOfSwapChainImages(); i++) {
00942         // descriptor write info
00943         VkWriteDescriptorSet descriptor_write{};
00944         descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00945         descriptor_write.dstSet = sharedRenderDescriptorSet[i];
00946         descriptor_write.dstBinding = TEXTURES_BINDING;
00947         descriptor_write.dstArrayElement = 0;
00948         descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00949         descriptor_write.descriptorCount =
00950             static_cast<uint32_t>(image_info_textures.size());
00951         descriptor_write.pImageInfo = image_info_textures.data();
00952
00953         /*VkDescriptorImageInfo sampler_info;
00954             sampler_info.imageView = nullptr;
00955             sampler_info.sampler = texture_sampler;*/
00956
00957         // descriptor write info
00958         VkWriteDescriptorSet descriptor_write_sampler{};
00959         descriptor_write_sampler.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00960         descriptor_write_sampler.dstSet = sharedRenderDescriptorSet[i];
00961         descriptor_write_sampler.dstBinding = SAMPLER_BINDING;
00962         descriptor_write_sampler.dstArrayElement = 0;
00963         descriptor_write_sampler.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
00964         descriptor_write_sampler.descriptorCount =
00965             static_cast<uint32_t>(image_info_texture_sampler.size());
00966         descriptor_write_sampler.pImageInfo = image_info_texture_sampler.data();
00967
00968         std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00969             descriptor_write, descriptor_write_sampler};
00970
00971         // update new descriptor set
00972         vkUpdateDescriptorSets(device->getLogicalDevice(),
00973             static_cast<uint32_t>(write_descriptor_sets.size()),
00974             write_descriptor_sets.data(), 0, nullptr);
00975     }
00976 }
```

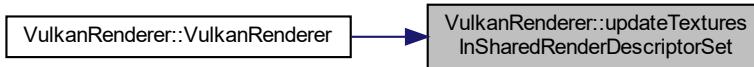
References [device](#), [VulkanSwapChain::getNumberOfSwapChainImages\(\)](#), [Scene::getTextureCount\(\)](#), [Scene::getTextures\(\)](#), [Scene::getTextureSampler\(\)](#), [scene](#), [sharedRenderDescriptorSet](#), and [vulkanSwapChain](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.43.3.29 updateUniforms()

```
void VulkanRenderer::updateUniforms (
    Scene * scene,
    Camera * camera,
    Window * window )
```

Definition at line 99 of file [VulkanRenderer.cpp](#).

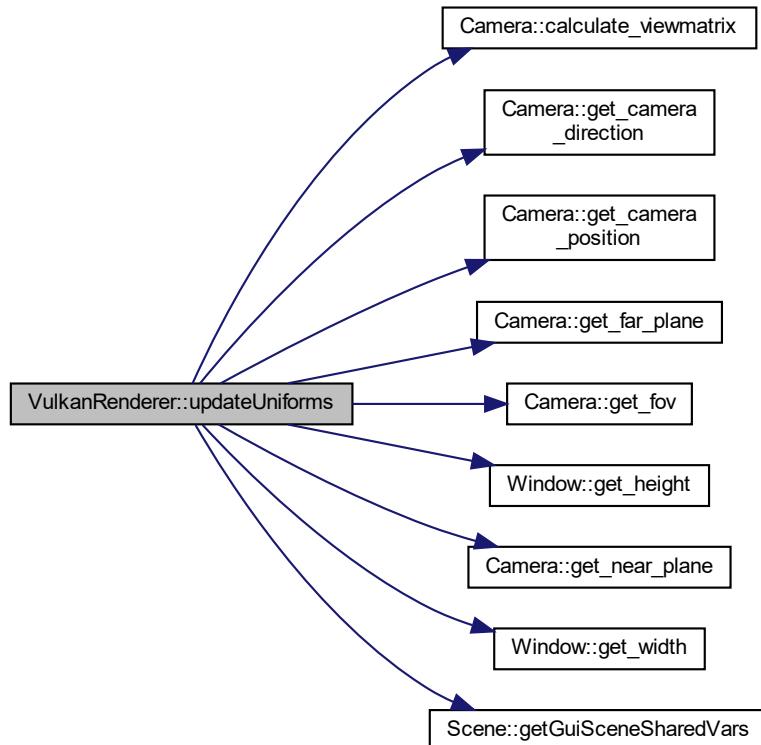
```

00100
00101     const GUISceneSharedVars guiSceneSharedVars = scene->getGuiSceneSharedVars();
00102
00103     globalUBO.view = camera->calculate_viewmatrix();
00104     globalUBO.projection =
00105         glm::perspective(glm::radians(camera->get_fov()),
00106                         (float)window->get_width() / (float)window->get_height(),
00107                         camera->get_near_plane(), camera->get_far_plane());
00108
00109     sceneUBO.view_dir = glm::vec4(camera->get_camera_direction(), 1.0f);
00110
00111     sceneUBO.light_dir =
00112         glm::vec4(guiSceneSharedVars.directional_light_direction[0],
00113                   guiSceneSharedVars.directional_light_direction[1],
00114                   guiSceneSharedVars.directional_light_direction[2], 1.0f);
00115
00116     sceneUBO.cam_pos =
00117         glm::vec4(camera->get_camera_position(), camera->get_fov());
00118 }
```

References [Camera::calculate\\_viewmatrix\(\)](#), [SceneUBO::cam\\_pos](#), [GUISceneSharedVars::directional\\_light\\_direction](#), [Camera::get\\_camera\\_direction\(\)](#), [Camera::get\\_camera\\_position\(\)](#), [Camera::get\\_far\\_plane\(\)](#), [Camera::get\\_fov\(\)](#), [Window::get\\_height\(\)](#), [Camera::get\\_near\\_plane\(\)](#), [Window::get\\_width\(\)](#), [Scene::getGuiSceneSharedVars\(\)](#), [globalUBO](#), [SceneUBO::light\\_dir](#), [GlobalUBO::projection](#), [scene](#), [sceneUBO](#), [GlobalUBO::view](#), [SceneUBO::view](#), [SceneUBO::view\\_dir](#), and [window](#).

Referenced by [VulkanRenderer\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.43.4 Field Documentation

#### 5.43.4.1 allocator

```
Allocator VulkanRenderer::allocator [private]
```

Definition at line 111 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.2 asManager

```
ASManager VulkanRenderer::asManager [private]
```

Definition at line 122 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [update\\_raytracing\\_descriptor\\_set\(\)](#), [updateRaytracingDescriptorSets\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.3 command\_buffers

```
std::vector<VkCommandBuffer> VulkanRenderer::command_buffers [private]
```

Definition at line 100 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [create\\_command\\_buffers\(\)](#), [drawFrame\(\)](#), [record\\_commands\(\)](#), and [update\\_uniform\\_buffers\(\)](#).

#### 5.43.4.4 commandBufferManager

```
CommandBufferManager VulkanRenderer::commandBufferManager [private]
```

Definition at line 101 of file [VulkanRenderer.hpp](#).

#### 5.43.4.5 compute\_command\_pool

```
VkCommandPool VulkanRenderer::compute_command_pool [private]
```

Definition at line 89 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUpCommandPools\(\)](#), and [create\\_command\\_pool\(\)](#).

#### 5.43.4.6 current\_frame

```
uint32_t VulkanRenderer::current_frame {0} [private]
```

Definition at line 114 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFrameBufferSize\(\)](#), and [drawFrame\(\)](#).

#### 5.43.4.7 descriptorPoolSharedRenderStages

```
VkDescriptorPool VulkanRenderer::descriptorPoolSharedRenderStages [private]
```

Definition at line 126 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [createDescriptorPoolSharedRenderStages\(\)](#), and [createSharedRenderDescriptorSet\(\)](#).

#### 5.43.4.8 device

```
std::unique_ptr<VulkanDevice> VulkanRenderer::device [private]
```

Definition at line 76 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFrameBufferSize\(\)](#), [cleanUp\(\)](#), [cleanUpCommandPools\(\)](#), [cleanUpSync\(\)](#), [create\\_command\\_buffers\(\)](#), [create\\_command\\_pool\(\)](#), [create\\_object\\_description\\_buffer\(\)](#), [create\\_post\\_descriptor\\_layout\(\)](#), [create\\_uniform\\_buffers\(\)](#), [createDescriptorPoolSharedRenderStages\(\)](#), [createRaytracingDescriptorPool\(\)](#), [createRaytracingDescriptorSetLayouts\(\)](#), [createRaytracingDescriptorSets\(\)](#), [createSharedRenderDescriptorSet\(\)](#), [createSharedRenderDescriptorSetLayouts\(\)](#), [createSynchronization\(\)](#), [drawFrame\(\)](#), [finishAllRenderCommands\(\)](#), [shaderHotReload\(\)](#), [update\\_raytracing\\_descriptor\\_set\(\)](#), [updatePostDescriptorSets\(\)](#), [updateRaytracingDescriptorSets\(\)](#), [updateTexturesInSharedRenderDescriptorSet\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.9 globalUBO

```
GlobalUBO VulkanRenderer::globalUBO [private]
```

Definition at line 92 of file [VulkanRenderer.hpp](#).

Referenced by [create\\_uniform\\_buffers\(\)](#), [createSharedRenderDescriptorSet\(\)](#), [update\\_uniform\\_buffers\(\)](#), and [updateUniforms\(\)](#).

#### 5.43.4.10 globalUBOBuffer

```
std::vector<VulkanBuffer> VulkanRenderer::globalUBOBuffer [private]
```

Definition at line 93 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUpUBOs\(\)](#), [create\\_uniform\\_buffers\(\)](#), [createDescriptorPoolSharedRenderStages\(\)](#), [createSharedRenderDescriptor\(\)](#) and [update\\_uniform\\_buffers\(\)](#).

#### 5.43.4.11 graphics\_command\_pool

```
VkCommandPool VulkanRenderer::graphics_command_pool [private]
```

Definition at line 88 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFramebufferSize\(\)](#), [cleanUp\(\)](#), [cleanUpCommandPools\(\)](#), [create\\_command\\_buffers\(\)](#), [create\\_command\\_pool\(\)](#), [create\\_object\\_description\\_buffer\(\)](#), [create\\_uniform\\_buffers\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.12 gui

```
GUI* VulkanRenderer::gui [private]
```

Definition at line 82 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFramebufferSize\(\)](#), [drawFrame\(\)](#), [record\\_commands\(\)](#), [updateStateDueToUserInput\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.13 image\_available

```
std::vector<VkSemaphore> VulkanRenderer::image_available [private]
```

Definition at line 115 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUpSync\(\)](#), [createSynchronization\(\)](#), and [drawFrame\(\)](#).

#### 5.43.4.14 images\_in\_flight\_fences

```
std::vector<VkFence> VulkanRenderer::images_in_flight_fences [private]
```

Definition at line 118 of file [VulkanRenderer.hpp](#).

Referenced by [createSynchronization\(\)](#), and [drawFrame\(\)](#).

#### 5.43.4.15 `in_flight_fences`

```
std::vector<VkFence> VulkanRenderer::in_flight_fences [private]
```

Definition at line 117 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUpSync\(\)](#), [createSynchronization\(\)](#), and [drawFrame\(\)](#).

#### 5.43.4.16 `instance`

```
VulkanInstance VulkanRenderer::instance [private]
```

Definition at line 69 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFramebufferSize\(\)](#), [cleanUp\(\)](#), [create\\_surface\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.17 `objectDescriptionBuffer`

```
VulkanBuffer VulkanRenderer::objectDescriptionBuffer [private]
```

Definition at line 123 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [create\\_object\\_description\\_buffer\(\)](#), and [update\\_raytracing\\_descriptor\\_set\(\)](#).

#### 5.43.4.18 `pathTracing`

```
PathTracing VulkanRenderer::pathTracing [private]
```

Definition at line 106 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [record\\_commands\(\)](#), [shaderHotReload\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.19 `post_descriptor_pool`

```
VkDescriptorPool VulkanRenderer::post_descriptor_pool {VK_NULL_HANDLE} [private]
```

Definition at line 134 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), and [create\\_post\\_descriptor\\_layout\(\)](#).

#### 5.43.4.20 post\_descriptor\_set

```
std::vector<VkDescriptorSet> VulkanRenderer::post_descriptor_set [private]
```

Definition at line 136 of file [VulkanRenderer.hpp](#).

Referenced by [create\\_post\\_descriptor\\_layout\(\)](#), [record\\_commands\(\)](#), and [updatePostDescriptorSets\(\)](#).

#### 5.43.4.21 post\_descriptor\_set\_layout

```
VkDescriptorSetLayout VulkanRenderer::post_descriptor_set_layout {VK_NULL_HANDLE} [private]
```

Definition at line 135 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFrameBufferSize\(\)](#), [cleanUp\(\)](#), [create\\_post\\_descriptor\\_layout\(\)](#), [shaderHotReload\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.22 postStage

```
PostStage VulkanRenderer::postStage [private]
```

Definition at line 107 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFrameBufferSize\(\)](#), [cleanUp\(\)](#), [record\\_commands\(\)](#), [shaderHotReload\(\)](#), [updatePostDescriptorSets\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.23 rasterizer

```
Rasterizer VulkanRenderer::rasterizer [private]
```

Definition at line 105 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFrameBufferSize\(\)](#), [cleanUp\(\)](#), [record\\_commands\(\)](#), [shaderHotReload\(\)](#), [updatePostDescriptorSets\(\)](#), [updateRaytracingDescriptorSets\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.24 raytracingDescriptorPool

```
VkDescriptorPool VulkanRenderer::raytracingDescriptorPool {VK_NULL_HANDLE} [private]
```

Definition at line 140 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [createRaytracingDescriptorPool\(\)](#), and [createRaytracingDescriptorSets\(\)](#).

#### 5.43.4.25 raytracingDescriptorSet

```
std::vector<VkDescriptorSet> VulkanRenderer::raytracingDescriptorSet [private]
```

Definition at line 141 of file [VulkanRenderer.hpp](#).

Referenced by [createRaytracingDescriptorSets\(\)](#), [record\\_commands\(\)](#), [update\\_raytracing\\_descriptor\\_set\(\)](#), and [updateRaytracingDescriptorSets\(\)](#).

#### 5.43.4.26 raytracingDescriptorsetLayout

```
VkDescriptorSetLayout VulkanRenderer::raytracingDescriptorsetLayout {VK_NULL_HANDLE} [private]
```

Definition at line 142 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [createRaytracingDescriptorSetLayouts\(\)](#), [createRaytracingDescriptorSets\(\)](#), [shaderHotReload\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.27 raytracingStage

```
Raytracing VulkanRenderer::raytracingStage [private]
```

Definition at line 104 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUp\(\)](#), [record\\_commands\(\)](#), [shaderHotReload\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.28 render\_finished

```
std::vector<VkSemaphore> VulkanRenderer::render_finished [private]
```

Definition at line 116 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUpSync\(\)](#), [createSynchronization\(\)](#), and [drawFrame\(\)](#).

#### 5.43.4.29 scene

```
Scene* VulkanRenderer::scene [private]
```

Definition at line 81 of file [VulkanRenderer.hpp](#).

Referenced by [create\\_object\\_description\\_buffer\(\)](#), [record\\_commands\(\)](#), [updateTexturesInSharedRenderDescriptorSet\(\)](#), [updateUniforms\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.30 sceneUBO

```
SceneUBO VulkanRenderer::sceneUBO [private]
```

Definition at line 94 of file [VulkanRenderer.hpp](#).

Referenced by [create\\_uniform\\_buffers\(\)](#), [createSharedRenderDescriptorSet\(\)](#), [update\\_uniform\\_buffers\(\)](#), and [updateUniforms\(\)](#).

#### 5.43.4.31 sceneUBOBuffer

```
std::vector<VulkanBuffer> VulkanRenderer::sceneUBOBuffer [private]
```

Definition at line 95 of file [VulkanRenderer.hpp](#).

Referenced by [cleanUpUBOs\(\)](#), [create\\_uniform\\_buffers\(\)](#), [createDescriptorPoolSharedRenderStages\(\)](#), [createSharedRenderDescriptor](#) and [update\\_uniform\\_buffers\(\)](#).

#### 5.43.4.32 sharedRenderDescriptorSet

```
std::vector<VkDescriptorSet> VulkanRenderer::sharedRenderDescriptorSet [private]
```

Definition at line 130 of file [VulkanRenderer.hpp](#).

Referenced by [create\\_object\\_description\\_buffer\(\)](#), [createSharedRenderDescriptorSet\(\)](#), [record\\_commands\(\)](#), [update\\_raytracing\\_descriptor\\_set\(\)](#), and [updateTexturesInSharedRenderDescriptorSet\(\)](#).

#### 5.43.4.33 sharedRenderDescriptorsetLayout

```
VkDescriptorSetLayout VulkanRenderer::sharedRenderDescriptorsetLayout [private]
```

Definition at line 128 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFramebufferSize\(\)](#), [cleanUp\(\)](#), [createSharedRenderDescriptorSet\(\)](#), [createSharedRenderDescriptorSet\(\)](#), [shaderHotReload\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.34 surface

```
VkSurfaceKHR VulkanRenderer::surface [private]
```

Definition at line 73 of file [VulkanRenderer.hpp](#).

Referenced by [checkChangedFramebufferSize\(\)](#), [cleanUp\(\)](#), [create\\_surface\(\)](#), and [VulkanRenderer\(\)](#).

#### 5.43.4.35 vulkanBufferManager

`VulkanBufferManager` `VulkanRenderer::vulkanBufferManager` [private]

Definition at line 66 of file [VulkanRenderer.hpp](#).

Referenced by `create_object_description_buffer()`, and `create_uniform_buffers()`.

#### 5.43.4.36 vulkanSwapChain

`VulkanSwapChain` `VulkanRenderer::vulkanSwapChain` [private]

Definition at line 78 of file [VulkanRenderer.hpp](#).

Referenced by `checkChangedFramebufferSize()`, `cleanUp()`, `create_command_buffers()`, `create_object_description_buffer()`, `create_post_descriptor_layout()`, `create_uniform_buffers()`, `createDescriptorPoolSharedRenderStages()`, `createRaytracingDescriptorPools()`, `createRaytracingDescriptorSets()`, `createSharedRenderDescriptorSet()`, `createSynchronization()`, `drawFrame()`, `record_commands()`, `updatePostDescriptorSets()`, `updateRaytracingDescriptorSets()`, `updateTexturesInSharedRenderDescriptorSet()` and `VulkanRenderer()`.

#### 5.43.4.37 window

`Window*` `VulkanRenderer::window` [private]

Definition at line 80 of file [VulkanRenderer.hpp](#).

Referenced by `checkChangedFramebufferSize()`, `create_surface()`, `updateUniforms()`, and `VulkanRenderer()`.

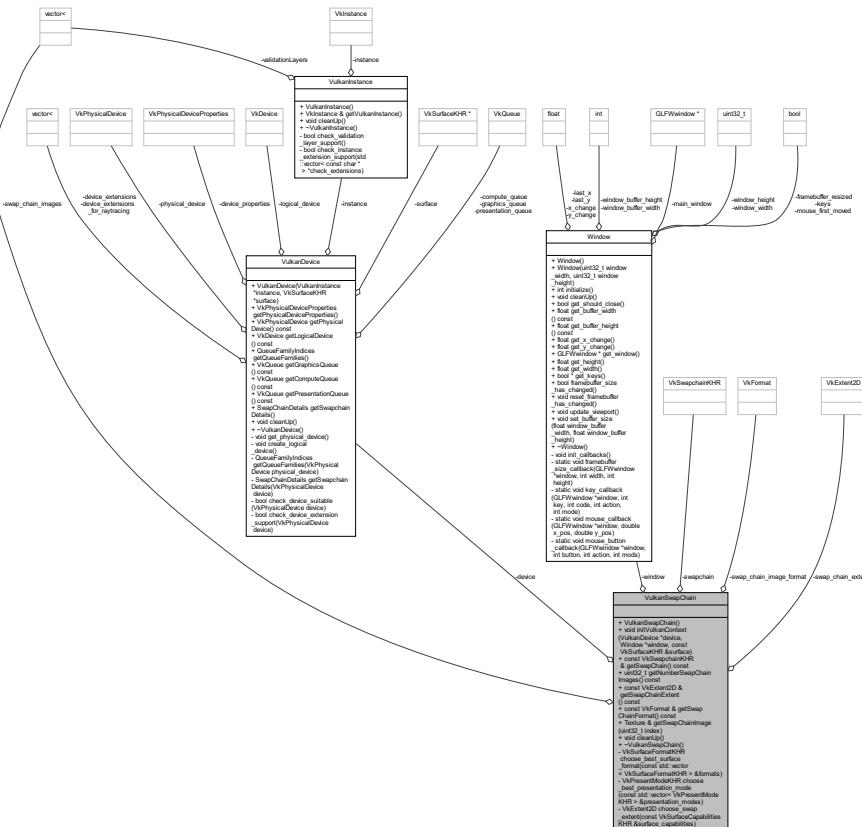
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/[VulkanRenderer.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/renderer/[VulkanRenderer.cpp](#)

## 5.44 VulkanSwapChain Class Reference

```
#include <VulkanSwapChain.hpp>
```

## Collaboration diagram for VulkanSwapChain:



## Public Member Functions

- `VulkanSwapChain ()`
  - `void initVulkanContext (VulkanDevice *device, Window *window, const VkSurfaceKHR &surface)`
  - `const VkSwapchainKHR & getSwapChain () const`
  - `uint32_t getNumberSwapChainImages () const`
  - `const VkExtent2D & getSwapChainExtent () const`
  - `const VkFormat & getSwapChainFormat () const`
  - `Texture & getSwapChainImage (uint32_t index)`
  - `void cleanUp ()`
  - `~VulkanSwapChain ()`

## Private Member Functions

- `VkSurfaceFormatKHR choose_best_surface_format (const std::vector< VkSurfaceFormatKHR > &formats)`
  - `VkPresentModeKHR choose_best_presentation_mode (const std::vector< VkPresentModeKHR > &presentation_modes)`
  - `VkExtent2D choose_swap_extent (const VkSurfaceCapabilitiesKHR &surface_capabilities)`

## Private Attributes

- `VulkanDevice * device {VK_NULL_HANDLE}`
- `Window * window {VK_NULL_HANDLE}`
- `VkSwapchainKHR swapchain {VK_NULL_HANDLE}`
- `std::vector< Texture > swap_chain_images`
- `VkFormat swap_chain_image_format {VK_FORMAT_B8G8R8A8_UNORM}`
- `VkExtent2D swap_chain_extent {0, 0}`

### 5.44.1 Detailed Description

Definition at line [8](#) of file [VulkanSwapChain.hpp](#).

### 5.44.2 Constructor & Destructor Documentation

#### 5.44.2.1 `VulkanSwapChain()`

`VulkanSwapChain::VulkanSwapChain ( )`

Definition at line [7](#) of file [VulkanSwapChain.cpp](#).  
00007 {}

#### 5.44.2.2 `~VulkanSwapChain()`

`VulkanSwapChain::~VulkanSwapChain ( )`

Definition at line [132](#) of file [VulkanSwapChain.cpp](#).  
00132 {}

### 5.44.3 Member Function Documentation

### 5.44.3.1 choose\_best\_presentation\_mode()

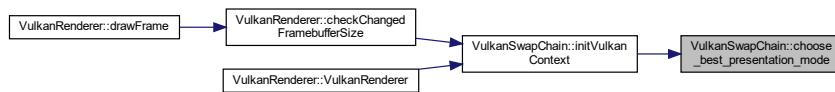
```
VkPresentModeKHR VulkanSwapChain::choose_best_presentation_mode (
    const std::vector< VkPresentModeKHR > & presentation_modes ) [private]
```

Definition at line 157 of file [VulkanSwapChain.cpp](#).

```
00158     {
00159         // look for mailbox presentation mode
00160         for (const auto& presentation_mode : presentation_modes) {
00161             if (presentation_mode == VK_PRESENT_MODE_MAILBOX_KHR) {
00162                 return presentation_mode;
00163             }
00164         }
00165
00166         // if can't find, use FIFO as Vulkan spec says it must be present
00167         return VK_PRESENT_MODE_FIFO_KHR;
00168     }
```

Referenced by [initVulkanContext\(\)](#).

Here is the caller graph for this function:



### 5.44.3.2 choose\_best\_surface\_format()

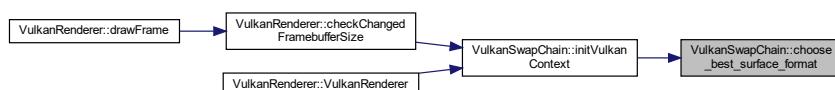
```
VkSurfaceFormatKHR VulkanSwapChain::choose_best_surface_format (
    const std::vector< VkSurfaceFormatKHR > & formats ) [private]
```

Definition at line 134 of file [VulkanSwapChain.cpp](#).

```
00135     {
00136         // best format is subjective, but I go with:
00137         // Format: VK_FORMAT_R8G8B8A8_UNORM (backup-format:
00138         // VK_FORMAT_B8G8R8A8_UNORM) color_space: VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
00139         // the condition in if means all formats are available (no restrictions)
00140         if (formats.size() == 1 && formats[0].format == VK_FORMAT_UNDEFINED) {
00141             return {VK_FORMAT_R8G8B8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR};
00142         }
00143
00144         // if restricted, search for optimal format
00145         for (const auto& format : formats) {
00146             if ((format.format == VK_FORMAT_R8G8B8A8_UNORM ||
00147                  format.format == VK_FORMAT_B8G8R8A8_UNORM) &&
00148                  format.colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
00149                 return format;
00150             }
00151         }
00152
00153         // in case just return first one--- but really shouldn't be the case ....
00154         return formats[0];
00155     }
```

Referenced by [initVulkanContext\(\)](#).

Here is the caller graph for this function:



### 5.44.3.3 choose\_swap\_extent()

```
VkExtent2D VulkanSwapChain::choose_swap_extent (
    const VkSurfaceCapabilitiesKHR & surface_capabilities ) [private]
```

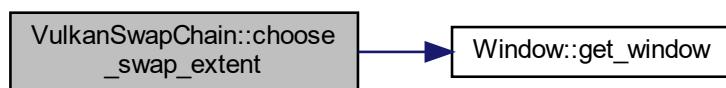
Definition at line 170 of file [VulkanSwapChain.cpp](#).

```
00171     {
00172     // if current extent is at numeric limits, than extent can vary. Otherwise it
00173     // is size of window
00174     if (surface_capabilities.currentExtent.width !=
00175         std::numeric_limits<uint32_t>::max()) {
00176         return surface_capabilities.currentExtent;
00177     } else {
00178         int width, height;
00179         glfwGetFramebufferSize(window->get_window(), &width, &height);
00180
00181         // create new extent using window size
00182         VkExtent2D new_extent{};
00183         new_extent.width = static_cast<uint32_t>(width);
00184         new_extent.height = static_cast<uint32_t>(height);
00185
00186         // surface also defines max and min, so make sure within boundaries bly
00187         // clamping value
00188         new_extent.width = std::max(
00189             surface_capabilities.minImageExtent.width,
00190             std::min(surface_capabilities.maxImageExtent.width, new_extent.width));
00191         new_extent.height =
00192             std::max(surface_capabilities.minImageExtent.height,
00193                     std::min(surface_capabilities.maxImageExtent.height,
00194                         new_extent.height));
00195
00196         return new_extent;
00197     }
00198 }
```

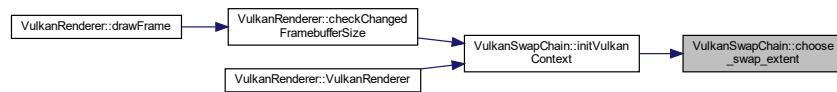
References [Window::get\\_window\(\)](#), and [window](#).

Referenced by [initVulkanContext\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.44.3.4 cleanUp()

```
void VulkanSwapChain::cleanUp ( )
```

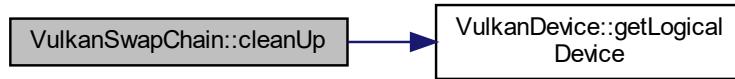
Definition at line 123 of file [VulkanSwapChain.cpp](#).

```
00123     {
00124         for (Texture& image : swap_chain_images) {
00125             vkDestroyImageView(device->getLogicalDevice(), image.getImageView(),
00126                                 nullptr);
00127         }
00128
00129         vkDestroySwapchainKHR(device->getLogicalDevice(), swapchain, nullptr);
00130     }
```

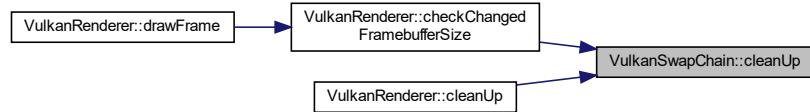
References [device](#), [VulkanDevice::getLogicalDevice\(\)](#), [swap\\_chain\\_images](#), and [swapchain](#).

Referenced by [VulkanRenderer::checkChangedFramebufferSize\(\)](#), and [VulkanRenderer::cleanUp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



#### 5.44.3.5 getNumberSwapChainImages()

```
uint32_t VulkanSwapChain::getNumberSwapChainImages ( ) const [inline]
```

Definition at line 16 of file [VulkanSwapChain.hpp](#).

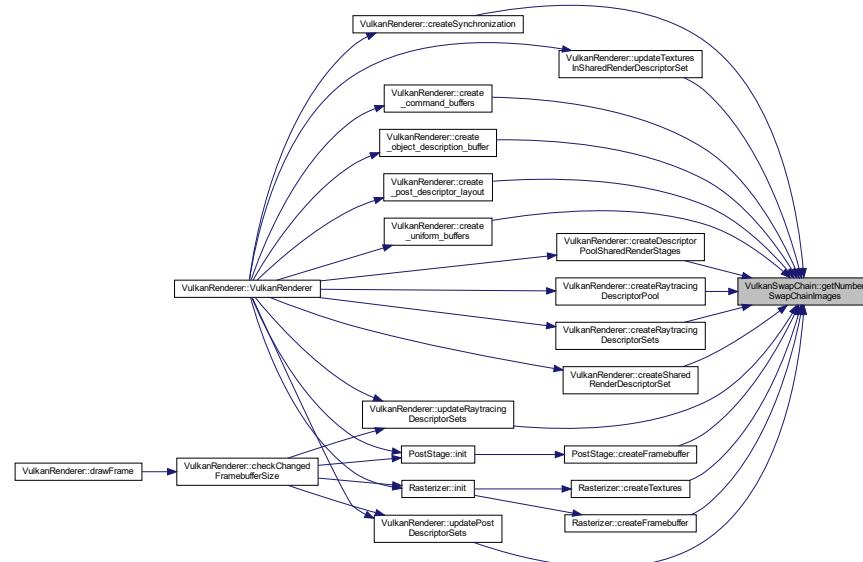
```
00016     {
00017         return static_cast<uint32_t>(swap_chain_images.size());
00018     };
```

References [swap\\_chain\\_images](#).

Referenced by [VulkanRenderer::create\\_command\\_buffers\(\)](#), [VulkanRenderer::create\\_object\\_description\\_buffer\(\)](#), [VulkanRenderer::create\\_post\\_descriptor\\_layout\(\)](#), [VulkanRenderer::create\\_uniform\\_buffers\(\)](#), [VulkanRenderer::createDescriptorPool\(\)](#), [PostStage::createFramebuffer\(\)](#), [Rasterizer::createFramebuffer\(\)](#), [VulkanRenderer::createRaytracingDescriptorPool\(\)](#),

[VulkanRenderer::createRaytracingDescriptorSets\(\)](#), [VulkanRenderer::createSharedRenderDescriptorSet\(\)](#), [VulkanRenderer::createSynchronization\(\)](#), [Rasterizer::createTextures\(\)](#), [VulkanRenderer::updatePostDescriptorSets\(\)](#), [VulkanRenderer::updateRaytracingDescriptorSets\(\)](#), and [VulkanRenderer::updateTexturesInSharedRenderDescriptorSet\(\)](#).

Here is the caller graph for this function:



#### 5.44.3.6 getSwapChain()

```
const VkSwapchainKHR & VulkanSwapChain::getSwapChain() const [inline]
```

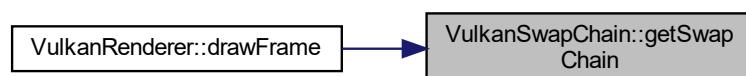
Definition at line 15 of file [VulkanSwapChain.hpp](#).

```
00015 { return swapchain; };
```

References [swapchain](#).

Referenced by [VulkanRenderer::drawFrame\(\)](#).

Here is the caller graph for this function:



### 5.44.3.7 getSwapChainExtent()

```
const VkExtent2D & VulkanSwapChain::getSwapChainExtent () const [inline]
```

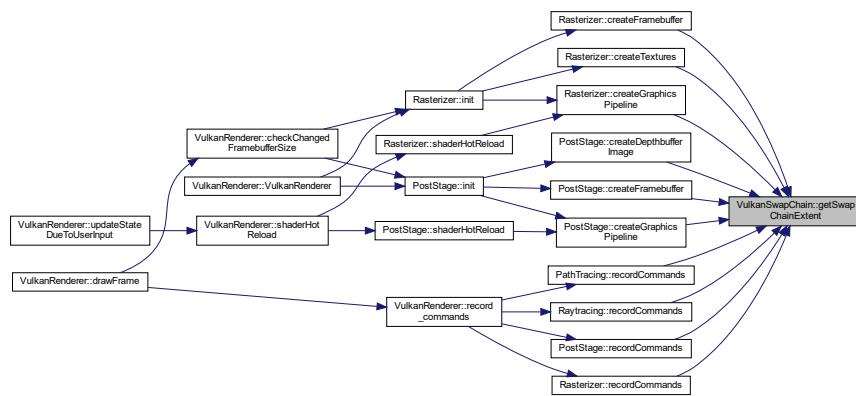
Definition at line 19 of file [VulkanSwapChain.hpp](#).

```
00019 { return swap_chain_extent; };
```

References [swap\\_chain\\_extent](#).

Referenced by [PostStage::createDepthbufferImage\(\)](#), [PostStage::createFramebuffer\(\)](#), [Rasterizer::createFramebuffer\(\)](#), [PostStage::createGraphicsPipeline\(\)](#), [Rasterizer::createGraphicsPipeline\(\)](#), [Rasterizer::createTextures\(\)](#), [PostStage::recordCommands\(\)](#), [Rasterizer::recordCommands\(\)](#), [PathTracing::recordCommands\(\)](#), and [Raytracing::recordCommands\(\)](#).

Here is the caller graph for this function:



### 5.44.3.8 getSwapChainFormat()

```
const VkFormat & VulkanSwapChain::getSwapChainFormat () const [inline]
```

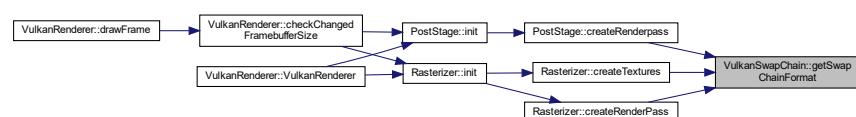
Definition at line 20 of file [VulkanSwapChain.hpp](#).

```
00020
00021     return swap_chain_image_format;
00022 }
```

References [swap\\_chain\\_image\\_format](#).

Referenced by [PostStage::createRenderpass\(\)](#), [Rasterizer::createRenderPass\(\)](#), and [Rasterizer::createTextures\(\)](#).

Here is the caller graph for this function:



### 5.44.3.9 getSwapChainImage()

```
Texture & VulkanSwapChain::getSwapChainImage (
    uint32_t index ) [inline]
```

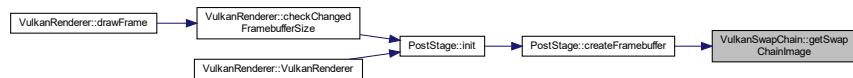
Definition at line 23 of file [VulkanSwapChain.hpp](#).

```
00023     return swap_chain_images[index];
00024 }
00025 }
```

References [swap\\_chain\\_images](#).

Referenced by [PostStage::createFramebuffer\(\)](#).

Here is the caller graph for this function:



### 5.44.3.10 initVulkanContext()

```
void VulkanSwapChain::initVulkanContext (
    VulkanDevice * device,
    Window * window,
    const VkSurfaceKHR & surface )
```

Definition at line 9 of file [VulkanSwapChain.cpp](#).

```
00010 {
00011     this->device = device;
00012     this->window = window;
00013
00014     // get swap chain details so we can pick the best settings
00015     SwapChainDetails swap_chain_details = device->getSwapchainDetails();
00016
00017     // 1. choose best surface format
00018     // 2. choose best presentation mode
00019     // 3. choose swap chain image resolution
00020
00021     VkSurfaceFormatKHR surface_format =
00022         choose_best_surface_format(swap_chain_details.formats);
00023     VkPresentModeKHR present_mode =
00024         choose_best_presentation_mode(swap_chain_details.presentation_mode);
00025     VkExtent2D extent =
00026         choose_swap_extent(swap_chain_details.surface_capabilities);
00027
00028     // how many images are in the swap chain; get 1 more than the minimum to allow
00029     // triple buffering
00030     uint32_t image_count =
00031         swap_chain_details.surface_capabilities.minImageCount + 1;
00032
00033     // if maxImageCount == 0, then limitless
00034     if (swap_chain_details.surface_capabilities.maxImageCount > 0 &&
00035         swap_chain_details.surface_capabilities.maxImageCount < image_count) {
00036         image_count = swap_chain_details.surface_capabilities.maxImageCount;
00037     }
00038
00039     VkSwapchainCreateInfoKHR swap_chain_create_info{};
00040     swap_chain_create_info.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
00041     swap_chain_create_info.surface = surface; // swapchain surface
00042     swap_chain_create_info.imageFormat =
00043         surface_format.format; // swapchain format
00044     swap_chain_create_info.imageColorSpace =
```

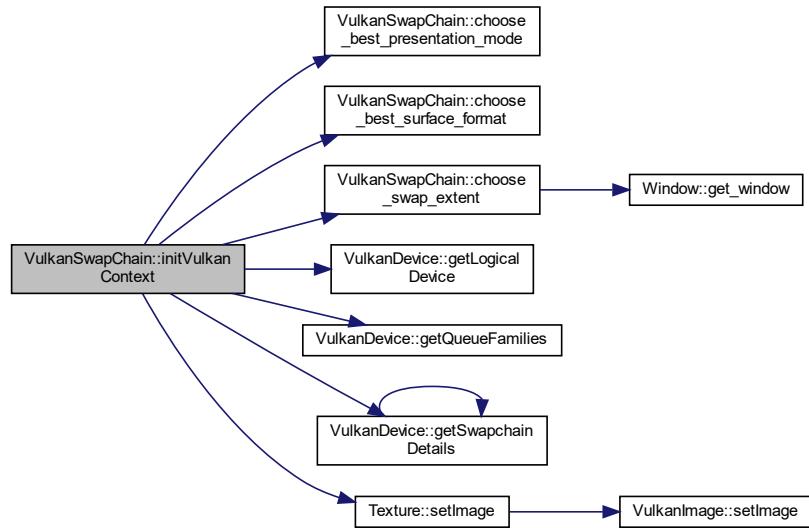
```

00045     surface_format.colorSpace; // swapchain color space
00046     swap_chain_create_info.presentMode =
00047         present_mode; // swapchain presentation mode
00048     swap_chain_create_info.imageExtent = extent; // swapchain image extents
00049     swap_chain_create_info.minImageCount =
00050         image_count; // minimum images in swapchain
00051     swap_chain_create_info.imageArrayLayers =
00052         1; // number of layers for each image in chain
00053     swap_chain_create_info.imageUsage =
00054         VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT |
00055         VK_IMAGE_USAGE_STORAGE_BIT |
00056         VK_IMAGE_USAGE_TRANSFER_DST_BIT; // what attachment images will be used
00057         // as
00058     swap_chain_create_info.preTransform =
00059         swap_chain_details.surface_capabilities
00060         .currentTransform; // transform to perform on swap chain images
00061     swap_chain_create_info.compositeAlpha =
00062         VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR; // dont do blending; everything opaque
00063     swap_chain_create_info.clipped = VK_TRUE; // of course activate clipping ! :)
00064
00065 // get queue family indices
00066 QueueFamilyIndices indices = device->getQueueFamilies();
00067
00068 // if graphics and presentation families are different then swapchain must let
00069 // images be shared between families
00070 if (indices.graphics_family != indices.presentation_family) {
00071     uint32_t queue_family_indices[] = {(uint32_t)indices.graphics_family,
00072                                         (uint32_t)indices.presentation_family};
00073
00074     swap_chain_create_info.imageSharingMode =
00075         VK_SHARING_MODE_CONCURRENT; // image share handling
00076     swap_chain_create_info.queueFamilyIndexCount =
00077         2; // number of queues to share images between
00078     swap_chain_create_info.pQueueFamilyIndices =
00079         queue_family_indices; // array of queues to share between
00080
00081 } else {
00082     swap_chain_create_info.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
00083     swap_chain_create_info.queueFamilyIndexCount = 0;
00084     swap_chain_create_info.pQueueFamilyIndices = nullptr;
00085 }
00086
00087 // if old swap chain been destroyed and this one replaces it then link old one
00088 // to quickly hand over responsibilities
00089 swap_chain_create_info.oldSwapchain = VK_NULL_HANDLE;
00090
00091 // create swap chain
00092 VkResult result = vkCreateSwapchainKHR(
00093     device->getLogicalDevice(), &swap_chain_create_info, nullptr, &swapchain);
00094 ASSERT_VULKAN(result, "Failed create swapchain!");
00095
00096 // store for later reference
00097 swap_chain_image_format = surface_format.format;
00098 swap_chain_extent = extent;
00099
00100 // get swapchain images (first count, then values)
00101 uint32_t swapchain_image_count;
00102 vkGetSwapchainImagesKHR(device->getLogicalDevice(), swapchain,
00103     &swapchain_image_count, nullptr);
00104 std::vector<VkImage> images(swapchain_image_count);
00105 vkGetSwapchainImagesKHR(device->getLogicalDevice(), swapchain,
00106     &swapchain_image_count, images.data());
00107
00108 swap_chain_images.clear();
00109
00110 for (size_t i = 0; i < images.size(); i++) {
00111     VkImage image = images[static_cast<uint32_t>(i)];
00112     // store image handle
00113     Texture swap_chain_image{};
00114     swap_chain_image.setImage(image);
00115     swap_chain_image.createImageView(device, swap_chain_image_format,
00116         VK_IMAGE_ASPECT_COLOR_BIT, 1);
00117
00118     // add to swapchain image list
00119     swap_chain_images.push_back(swap_chain_image);
00120 }
00121 }
```

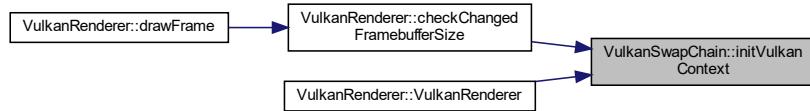
References `choose_best_presentation_mode()`, `choose_best_surface_format()`, `choose_swap_extent()`, `device`, `SwapChainDetails::formats`, `VulkanDevice::getLogicalDevice()`, `VulkanDevice::getQueueFamilies()`, `VulkanDevice::getSwapchainDetails()`, `QueueFamilyIndices::graphics_family`, `QueueFamilyIndices::presentation_family`, `SwapChainDetails::presentation_mode`, `Texture::setImage()`, `SwapChainDetails::surface_capabilities`, `swap_chain_extent`, `swap_chain_image_format`, `swap_chain_images`, `swapchain`, and `window`.

Referenced by `VulkanRenderer::checkChangedFramebufferSize()`, and `VulkanRenderer::VulkanRenderer()`.

Here is the call graph for this function:



Here is the caller graph for this function:



## 5.44.4 Field Documentation

### 5.44.4.1 device

`VulkanDevice* VulkanSwapChain::device {VK_NULL_HANDLE} [private]`

Definition at line 32 of file [VulkanSwapChain.hpp](#).

Referenced by [cleanUp\(\)](#), and [initVulkanContext\(\)](#).

#### 5.44.4.2 swap\_chain\_extent

```
VkExtent2D VulkanSwapChain::swap_chain_extent {0, 0} [private]
```

Definition at line 39 of file [VulkanSwapChain.hpp](#).

Referenced by [getSwapChainExtent\(\)](#), and [initVulkanContext\(\)](#).

#### 5.44.4.3 swap\_chain\_image\_format

```
VkFormat VulkanSwapChain::swap_chain_image_format {VK_FORMAT_B8G8R8A8_UNORM} [private]
```

Definition at line 38 of file [VulkanSwapChain.hpp](#).

Referenced by [getSwapChainFormat\(\)](#), and [initVulkanContext\(\)](#).

#### 5.44.4.4 swap\_chain\_images

```
std::vector<Texture> VulkanSwapChain::swap_chain_images [private]
```

Definition at line 37 of file [VulkanSwapChain.hpp](#).

Referenced by [cleanUp\(\)](#), [getNumberSwapChainImages\(\)](#), [getSwapChainImage\(\)](#), and [initVulkanContext\(\)](#).

#### 5.44.4.5 swapchain

```
VkSwapchainKHR VulkanSwapChain::swapchain {VK_NULL_HANDLE} [private]
```

Definition at line 35 of file [VulkanSwapChain.hpp](#).

Referenced by [cleanUp\(\)](#), [getSwapChain\(\)](#), and [initVulkanContext\(\)](#).

#### 5.44.4.6 window

```
Window* VulkanSwapChain::window {VK_NULL_HANDLE} [private]
```

Definition at line 33 of file [VulkanSwapChain.hpp](#).

Referenced by [choose\\_swap\\_extent\(\)](#), and [initVulkanContext\(\)](#).

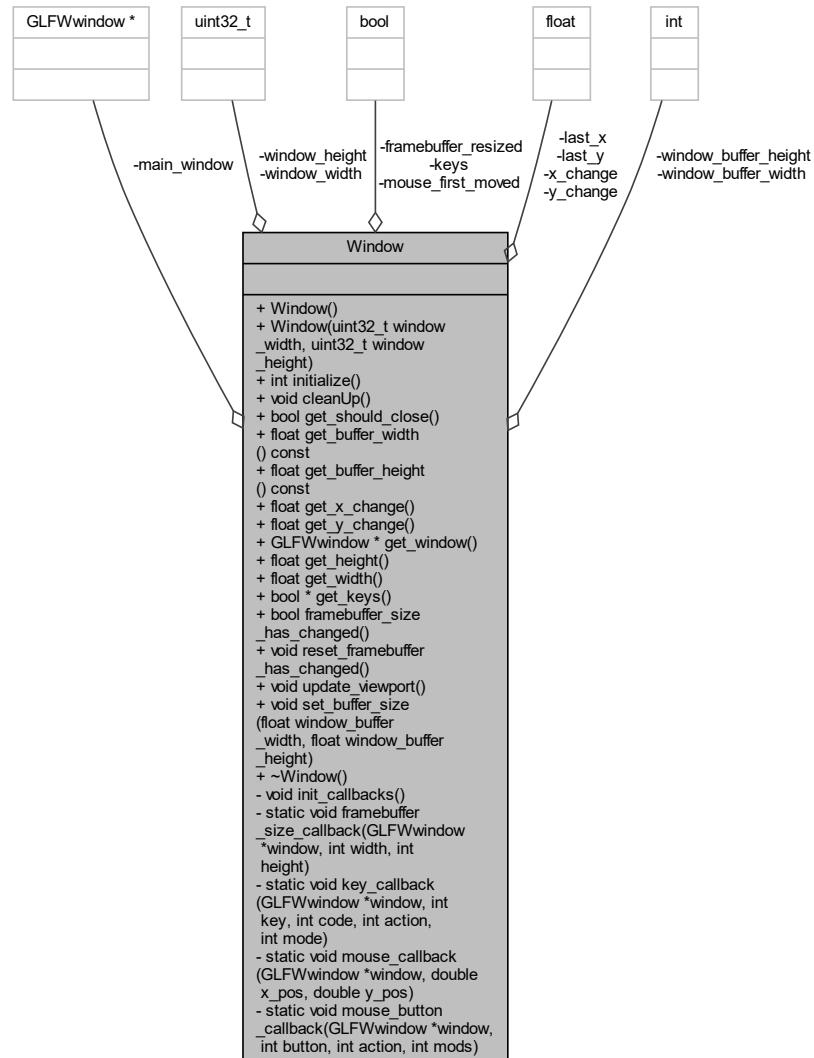
The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/[VulkanSwapChain.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/[VulkanSwapChain.cpp](#)

## 5.45 Window Class Reference

```
#include <Window.hpp>
```

Collaboration diagram for Window:



### Public Member Functions

- [Window \(\)](#)
- [Window \(uint32\\_t window\\_width, uint32\\_t window\\_height\)](#)
- [int initialize \(\)](#)
- [void cleanUp \(\)](#)
- [bool get\\_should\\_close \(\)](#)
- [float get\\_buffer\\_width \(\) const](#)
- [float get\\_buffer\\_height \(\) const](#)
- [float get\\_x\\_change \(\)](#)

- float `get_y_change ()`
- GLFWwindow \* `get_window ()`
- float `get_height ()`
- float `get_width ()`
- bool \* `get_keys ()`
- bool `framebuffer_size_has_changed ()`
- void `reset_framebuffer_has_changed ()`
- void `update_viewport ()`
- void `set_buffer_size (float window_buffer_width, float window_buffer_height)`
- `~Window ()`

## Private Member Functions

- void `init_callbacks ()`

## Static Private Member Functions

- static void `framebuffer_size_callback (GLFWwindow *window, int width, int height)`
- static void `key_callback (GLFWwindow *window, int key, int code, int action, int mode)`
- static void `mouse_callback (GLFWwindow *window, double x_pos, double y_pos)`
- static void `mouse_button_callback (GLFWwindow *window, int button, int action, int mods)`

## Private Attributes

- GLFWwindow \* `main_window`
- uint32\_t `window_width`
- uint32\_t `window_height`
- bool `keys [1024]`
- float `last_x`
- float `last_y`
- float `x_change`
- float `y_change`
- bool `mouse_first_moved`
- bool `framebuffer_resized`
- int `window_buffer_width`
- int `window_buffer_height`

### 5.45.1 Detailed Description

Definition at line 9 of file `Window.hpp`.

### 5.45.2 Constructor & Destructor Documentation

### 5.45.2.1 Window() [1/2]

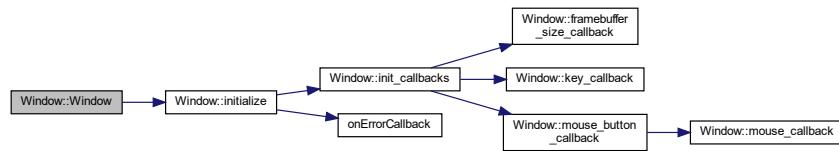
```
Window::Window ( )
```

Definition at line 14 of file [Window.cpp](#).

```
00015   :
00016
00017   window_width(800.f),
00018   window_height(600.f),
00019   x_change(0.0f),
00020   y_change(0.0f),
00021   framebuffer_resized(false)
00022
00023 {
00024   // all keys non-pressed in the beginning
00025   for (size_t i = 0; i < 1024; i++) {
00026     keys[i] = 0;
00027   }
00028   initialize();
00030 }
```

References [initialize\(\)](#), and [keys](#).

Here is the call graph for this function:



### 5.45.2.2 Window() [2/2]

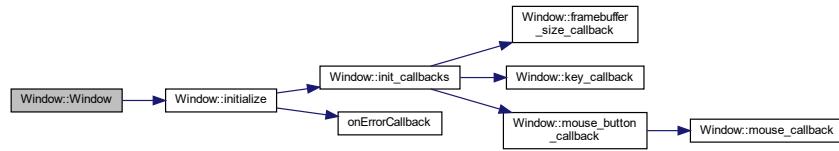
```
Window::Window (
    uint32_t window_width,
    uint32_t window_height )
```

Definition at line 33 of file [Window.cpp](#).

```
00034   :
00035
00036   window_width(window_width),
00037   window_height(window_height),
00038   x_change(0.0f),
00039   y_change(0.0f),
00040   framebuffer_resized(false)
00041
00042 {
00043   // all keys non-pressed in the beginning
00044   for (size_t i = 0; i < 1024; i++) {
00045     keys[i] = 0;
00046   }
00047   initialize();
00048 }
```

References [initialize\(\)](#), and [keys](#).

Here is the call graph for this function:



### 5.45.2.3 ~Window()

```
Window::~Window ( )
```

Definition at line 198 of file [Window.cpp](#).

```
00198 { }
```

## 5.45.3 Member Function Documentation

### 5.45.3.1 cleanUp()

```
void Window::cleanUp ( )
```

Definition at line 87 of file [Window.cpp](#).

```
00087     {
00088     glfwDestroyWindow(main_window);
00089     glfwTerminate();
00090 }
```

References [main\\_window](#).

### 5.45.3.2 framebuffer\_size\_callback()

```
void Window::framebuffer_size_callback (
    GLFWwindow * window,
    int width,
    int height ) [static], [private]
```

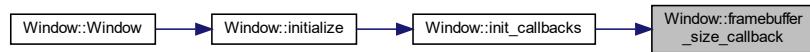
Definition at line 130 of file [Window.cpp](#).

```
00131     {
00132     auto app = reinterpret_cast<Window*>(glfwGetWindowUserPointer(window));
00133     app->framebuffer_resized = true;
00134     app->window_width = width;
00135     app->window_height = height;
00136 }
```

References [framebuffer\\_resized](#).

Referenced by [init\\_callbacks\(\)](#).

Here is the caller graph for this function:



#### 5.45.3.3 framebuffer\_size\_has\_changed()

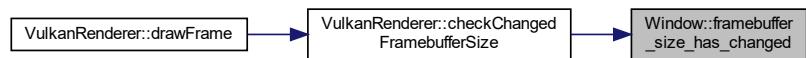
```
bool Window::framebuffer_size_has_changed ( )
```

Definition at line 119 of file [Window.cpp](#).  
00119 { [return framebuffer\\_resized](#); }

References [framebuffer\\_resized](#).

Referenced by [VulkanRenderer::checkChangedFrameBufferSize\(\)](#).

Here is the caller graph for this function:



#### 5.45.3.4 get\_buffer\_height()

```
float Window::get_buffer_height ( ) const [inline]
```

Definition at line 21 of file [Window.hpp](#).  
00021 { [return \(float\)window\\_buffer\\_height](#); }

References [window\\_buffer\\_height](#).

### 5.45.3.5 get\_buffer\_width()

```
float Window::get_buffer_width ( ) const [inline]
```

Definition at line 20 of file [Window.hpp](#).

```
00020 { return (float)window_buffer_width; }
```

References [window\\_buffer\\_width](#).

### 5.45.3.6 get\_height()

```
float Window::get_height ( )
```

Definition at line 115 of file [Window.cpp](#).

```
00115 { return float(window_height); }
```

References [window\\_height](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.45.3.7 get\_keys()

```
bool * Window::get_keys ( ) [inline]
```

Definition at line 29 of file [Window.hpp](#).

```
00029 { return keys; }
```

References [keys](#).

### 5.45.3.8 get\_should\_close()

```
bool Window::get_should_close ( ) [inline]
```

Definition at line 19 of file [Window.hpp](#).

```
00019 { return glfwWindowShouldClose(main_window); }
```

References [main\\_window](#).

### 5.45.3.9 get\_width()

```
float Window::get_width ( )
```

Definition at line 117 of file [Window.cpp](#).  
 00117 { **return** float([window\\_width](#)); }

References [window\\_width](#).

Referenced by [VulkanRenderer::updateUniforms\(\)](#).

Here is the caller graph for this function:



### 5.45.3.10 get\_window()

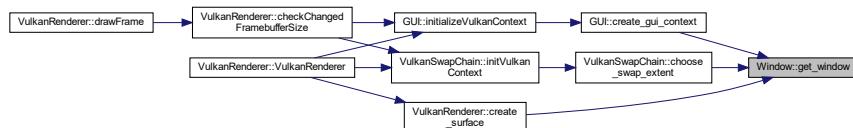
```
GLFWwindow * Window::get_window ( ) [inline]
```

Definition at line 24 of file [Window.hpp](#).  
 00024 { **return** [main\\_window](#); }

References [main\\_window](#).

Referenced by [VulkanSwapChain::choose\\_swap\\_extent\(\)](#), [GUI::create\\_gui\\_context\(\)](#), and [VulkanRenderer::create\\_surface\(\)](#).

Here is the caller graph for this function:



### 5.45.3.11 get\_x\_change()

```
float Window::get_x_change ( )
```

Definition at line 103 of file [Window.cpp](#).  
 00103 {
 00104 float the\_change = [x\\_change](#);
 00105 [x\\_change](#) = 0.0f;
 00106 **return** the\_change;
 00107 }

References [x\\_change](#).

### 5.45.3.12 get\_y\_change()

```
float Window::get_y_change ( )
```

Definition at line 109 of file [Window.cpp](#).

```
00109     {
00110     float the_change = y_change;
00111     y_change = 0.0f;
00112     return the_change;
00113 }
```

References [y\\_change](#).

### 5.45.3.13 init\_callbacks()

```
void Window::init_callbacks ( ) [private]
```

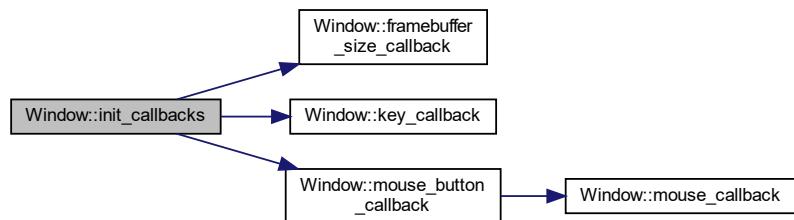
Definition at line 121 of file [Window.cpp](#).

```
00121     {
00122     // TODO: remember this section for our later game logic
00123     // for the space ship to fly around
00124     glfwSetWindowUserPointer(main_window, this);
00125     glfwSetKeyCallback(main_window, &key_callback);
00126     glfwSetMouseButtonCallback(main_window, &mouse_button_callback);
00127     glfwSetFramebufferSizeCallback(main_window, &framebuffer_size_callback);
00128 }
```

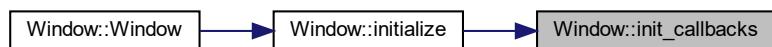
References [framebuffer\\_size\\_callback\(\)](#), [key\\_callback\(\)](#), [main\\_window](#), and [mouse\\_button\\_callback\(\)](#).

Referenced by [initialize\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.45.3.14 initialize()

```
int Window::initialize ( )
```

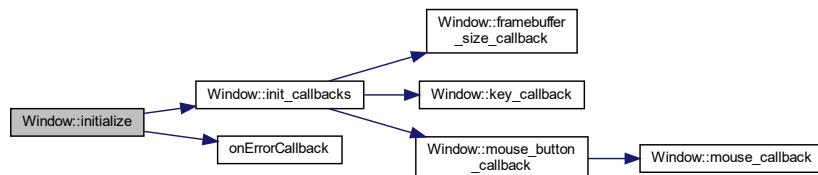
Definition at line 51 of file [Window.cpp](#).

```
00051     {
00052     glfwSetErrorCallback(onErrorCallback);
00053     if (!glfwInit()) {
00054         printf("GLFW Init failed!");
00055         glfwTerminate();
00056         return 1;
00057     }
00058
00059     if (!glfwVulkanSupported()) {
00060         throw std::runtime_error("No Vulkan Supported!");
00061     }
00062
00063     // allow it to resize
00064     glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
00065
00066     // retrieve new window
00067     glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
00068     main_window =
00069         glfwCreateWindow(window_width, window_height,
00070                         "\\\\_ Epic graphics from hell \\_\_ ", NULL, NULL);
00071
00072     if (!main_window) {
00073         printf("GLFW Window creation failed!");
00074         glfwTerminate();
00075         return 1;
00076     }
00077
00078     // get buffer size information
00079     glfwGetFramebufferSize(main_window, &window_buffer_width,
00080                           &window_buffer_height);
00081
00082     init_callbacks();
00083
00084     return 0;
00085 }
```

References [init\\_callbacks\(\)](#), [main\\_window](#), [onErrorCallback\(\)](#), [window\\_buffer\\_height](#), [window\\_buffer\\_width](#), [window\\_height](#), and [window\\_width](#).

Referenced by [Window\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.45.3.15 key\_callback()

```
void Window::key_callback (
    GLFWwindow * window,
    int key,
    int code,
    int action,
    int mode ) [static], [private]
```

Definition at line 142 of file [Window.cpp](#).

```
00143     {
00144     Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));
00145
00146     if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) {
00147         glfwSetWindowShouldClose(window, VK_TRUE);
00148     }
00149
00150     if (key >= 0 && key < 1024) {
00151         if (action == GLFW_PRESS) {
00152             the_window->keys[key] = true;
00153
00154         } else if (action == GLFW_RELEASE) {
00155             the_window->keys[key] = false;
00156         }
00157     }
00158 }
```

References [keys](#).

Referenced by [init\\_callbacks\(\)](#).

Here is the caller graph for this function:



### 5.45.3.16 mouse\_button\_callback()

```
void Window::mouse_button_callback (
    GLFWwindow * window,
    int button,
    int action,
    int mods ) [static], [private]
```

Definition at line 179 of file [Window.cpp](#).

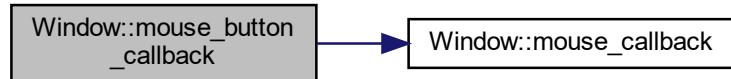
```
00180     {
00181     if (ImGui::GetCurrentContext() != nullptr &&
00182         ImGui::GetIO().WantCaptureMouse) {
00183         ImGuiIO& io = ImGui::GetIO();
00184         io.AddMouseButtonEvent(button, action);
00185         return;
00186     }
00187
00188     Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));
00189
00190     if ((action == GLFW_PRESS) && (button == GLFW_MOUSE_BUTTON_RIGHT)) {
00191         glfwSetCursorPosCallback(window, mouse_callback);
00192     } else {
00193         the_window->mouse_first_moved = true;
00194         glfwSetCursorPosCallback(window, NULL);
  
```

```
00195 }
00196 }
```

References [mouse\\_callback\(\)](#), and [mouse\\_first\\_moved](#).

Referenced by [init\\_callbacks\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### 5.45.3.17 mouse\_callback()

```
void Window::mouse_callback (
    GLFWwindow * window,
    double x_pos,
    double y_pos ) [static], [private]
```

Definition at line 160 of file [Window.cpp](#).

```
00160
00161     Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));
00162
00163     // need to handle first occurrence of a mouse moving event
00164     if (the_window->mouse_first_moved) {
00165         the_window->last_x = static_cast<float>(x_pos);
00166         the_window->last_y = static_cast<float>(y_pos);
00167         the_window->mouse_first_moved = false;
00168     }
00169
00170     the_window->x_change = static_cast<float>((x_pos - the_window->last_x));
00171     // take care of correct subtraction :
00172     the_window->y_change = static_cast<float>((the_window->last_y - y_pos));
00173
00174     // update params
00175     the_window->last_x = static_cast<float>(x_pos);
00176     the_window->last_y = static_cast<float>(y_pos);
00177 }
```

References [last\\_x](#), [last\\_y](#), [mouse\\_first\\_moved](#), [x\\_change](#), and [y\\_change](#).

Referenced by [mouse\\_button\\_callback\(\)](#).

Here is the caller graph for this function:



#### 5.45.3.18 reset\_framebuffer\_has\_changed()

```
void Window::reset_framebuffer_has_changed( )
```

Definition at line 138 of file [Window.cpp](#).

```
00138                                     {
00139     this->framebuffer_resized = false;
00140 }
```

References [framebuffer\\_resized](#).

Referenced by [VulkanRenderer::checkChangedFrameBufferSize\(\)](#).

Here is the caller graph for this function:



#### 5.45.3.19 set\_buffer\_size()

```
void Window::set_buffer_size(
    float window_buffer_width,
    float window_buffer_height )
```

Definition at line 97 of file [Window.cpp](#).

```
00098                                     {
00099     this->window_buffer_width = window_buffer_width;
00100     this->window_buffer_height = window_buffer_height;
00101 }
```

References [window\\_buffer\\_height](#), and [window\\_buffer\\_width](#).

### 5.45.3.20 update\_viewport()

```
void Window::update_viewport ( )
```

Definition at line 92 of file [Window.cpp](#).

```
00092     {
00093     glfwGetFramebufferSize(main\_window, &window\_buffer\_width,
00094                           &window\_buffer\_height);
00095 }
```

References [main\\_window](#), [window\\_buffer\\_height](#), and [window\\_buffer\\_width](#).

## 5.45.4 Field Documentation

### 5.45.4.1 framebuffer\_resized

```
bool Window::framebuffer_resized [private]
```

Definition at line 49 of file [Window.hpp](#).

Referenced by [framebuffer\\_size\\_callback\(\)](#), [framebuffer\\_size\\_has\\_changed\(\)](#), and [reset\\_framebuffer\\_has\\_changed\(\)](#).

### 5.45.4.2 keys

```
bool Window::keys[1024] [private]
```

Definition at line 43 of file [Window.hpp](#).

Referenced by [get\\_keys\(\)](#), [key\\_callback\(\)](#), and [Window\(\)](#).

### 5.45.4.3 last\_x

```
float Window::last_x [private]
```

Definition at line 44 of file [Window.hpp](#).

Referenced by [mouse\\_callback\(\)](#).

#### 5.45.4.4 `last_y`

```
float Window::last_y [private]
```

Definition at line [45](#) of file [Window.hpp](#).

Referenced by [mouse\\_callback\(\)](#).

#### 5.45.4.5 `main_window`

```
GLFWwindow* Window::main_window [private]
```

Definition at line [40](#) of file [Window.hpp](#).

Referenced by [cleanUp\(\)](#), [get\\_should\\_close\(\)](#), [get\\_window\(\)](#), [init\\_callbacks\(\)](#), [initialize\(\)](#), and [update\\_viewport\(\)](#).

#### 5.45.4.6 `mouse_first_moved`

```
bool Window::mouse_first_moved [private]
```

Definition at line [48](#) of file [Window.hpp](#).

Referenced by [mouse\\_button\\_callback\(\)](#), and [mouse\\_callback\(\)](#).

#### 5.45.4.7 `window_buffer_height`

```
int Window::window_buffer_height [private]
```

Definition at line [52](#) of file [Window.hpp](#).

Referenced by [get\\_buffer\\_height\(\)](#), [initialize\(\)](#), [set\\_buffer\\_size\(\)](#), and [update\\_viewport\(\)](#).

#### 5.45.4.8 `window_buffer_width`

```
int Window::window_buffer_width [private]
```

Definition at line [52](#) of file [Window.hpp](#).

Referenced by [get\\_buffer\\_width\(\)](#), [initialize\(\)](#), [set\\_buffer\\_size\(\)](#), and [update\\_viewport\(\)](#).

#### 5.45.4.9 window\_height

```
uint32_t Window::window_height [private]
```

Definition at line 41 of file [Window.hpp](#).

Referenced by [get\\_height\(\)](#), and [initialize\(\)](#).

#### 5.45.4.10 window\_width

```
uint32_t Window::window_width [private]
```

Definition at line 41 of file [Window.hpp](#).

Referenced by [get\\_width\(\)](#), and [initialize\(\)](#).

#### 5.45.4.11 x\_change

```
float Window::x_change [private]
```

Definition at line 46 of file [Window.hpp](#).

Referenced by [get\\_x\\_change\(\)](#), and [mouse\\_callback\(\)](#).

#### 5.45.4.12 y\_change

```
float Window::y_change [private]
```

Definition at line 47 of file [Window.hpp](#).

Referenced by [get\\_y\\_change\(\)](#), and [mouse\\_callback\(\)](#).

The documentation for this class was generated from the following files:

- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/window/[Window.hpp](#)
- C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/window/[Window.cpp](#)



# Chapter 6

## File Documentation

### 6.1 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/CompilerWarnings.cmake File Reference

### 6.2 CompilerWarnings.cmake

[Go to the documentation of this file.](#)

```
00001 function(set_project_warnings project_name)
00002     option(TREAT_WARNINGS_AS_ERRORS "Treat warnings as errors" OFF)
00003     # add compile flags; as restrictive as possible
00004     # for windows
00005     # source:
00006     https://github.com/cpp-best-practices/cppbestpractices/blob/master/02-Use_the_Tools_Available.md
00007     set(MSVC_WARNINGS
00008         /W4 # All reasonable warnings
00009         /w14242 #'identifier': conversion from 'type1' to 'type1', possible loss of data
00010         /w14254 #'operator': conversion from 'type1:field_bits' to 'type2:field_bits', possible
00011         loss of data
00012         /w14263 #'function': member function does not override any base class virtual member
00013         function
00014         /w14265 #'classname': class has virtual functions, but destructor is not virtual
00015         instances of this class may not be destructed correctly
00016         /w14287 #'operator': unsigned/negative constant mismatch
00017         /we4289 #nonstandard extension used: 'variable': loop control variable declared in the
00018         for-loop is used outside the for-loop scope
00019         /w14296 #'operator': expression is always 'boolean_value'
00020         /w14311 #'variable': pointer truncation from 'type1' to 'type2'
00021         /w14545 #expression before comma evaluates to a function which is missing an argument
00022         list
00023         /w14546 #function call before comma missing argument list
00024         /w14547 #'operator': operator before comma has no effect; expected operator with
00025         side-effect
00026         /w14549 #'operator': operator before comma has no effect; did you intend 'operator'?
00027         /w14555 #expression has no effect; expected expression with side-effect
00028         /w14619 #pragma warning: there is no warning number 'number'
00029         /w14640 #Enable warning on thread un-safe static member initialization
00030         /w14826 #Conversion from 'type1' to 'type_2' is sign-extended. This may cause unexpected
00031         runtime behavior.
00032         /w14905 #wide string literal cast to 'LPSTR'
00033         /w14906 #string literal cast to 'LPWSTR'
00034         /w14928 #illegal copy-initialization; more than one user-defined conversion has been
00035         implicitly applied
00036     )
00037
00038     # CLANG
00039     set(CLANG_WARNINGS
00040         -Wall -Wextra # reasonable and standard
00041         -Wshadow # warn the user if a variable declaration shadows one from a parent context
00042         -Wnon-virtual-dtor # warn the user if a class with virtual functions has a non-virtual
00043         destructor. This helps catch hard to track down memory errors
00044         -Wold-style-cast # warn for c-style casts
00045         -Wcast-align # warn for potential performance problem casts
00046         -Wunused # warn on anything being unused
00047         -Woverloaded-virtual # warn if you overload (not override) a virtual function
```

```

00038      -Wpedantic # (all versions of GCC, Clang >= 3.2) warn if non-standard C++ is used
00039      -Wconversion # warn on type conversions that may lose data
00040      -Wsighn-conversion # (Clang all versions, GCC >= 4.3) warn on sign conversions
00041      -Wnull-dereference # (only in GCC >= 6.0) warn if a null dereference is detected
00042      -Wdouble-promotion # (GCC >= 4.6, Clang >= 3.8) warn if float is implicit promoted to
00043      double           -Wformat=2 # warn on security issues around functions that format output (ie printf)
00044  )
00045
00046  if(TREAT_WARNINGS_AS_ERRORS)
00047      set(CLANG_WARNINGS ${CLANG_WARNINGS} -Werror)
00048      set(MSVC_WARNINGS ${MSVC_WARNINGS} /WX)
00049  endif()
00050
00051  # GCC
00052  set(GCC_WARNINGS ${CLANG_WARNINGS}
00053      -Wmisleading-indentation # (only in GCC >= 6.0) warn if indentation implies
00054  blocks where blocks do not exist)
00055  duplicated conditions
00056  duplicated code
00057  bitwise were probably wanted
00058  type
00059  )
00060
00061  if(MSVC)
00062      set(PROJECT_WARNINGS ${MSVC_WARNINGS})
00063  elseif(CMAKE_CXX_COMPILER_ID STREQUAL "Clang")
00064      set(PROJECT_WARNINGS ${CLANG_WARNINGS})
00065  else()
00066      set(PROJECT_WARNINGS ${GCC_WARNINGS})
00067  endif()
00068  target_compile_options(${project_name} INTERFACE ${PROJECT_WARNINGS})
00069
00070 endfunction()

```

## 6.3 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/← CompileShadersToSPV.cmake File Reference

### 6.4 CompileShadersToSPV.cmake

[Go to the documentation of this file.](#)

```

00001 # compile glslc shaders
00002 # source:
00003     https://www.reddit.com/r/vulkan/comments/kbaxlz/what_is_your_workflow_when_compiling_shader_files/
00004 function(add_shader TARGET SHADER)
00005  find_program(GLSLC glslc)
00006  set(current-shader-path ${SHADER}) #${CMAKE_CURRENT_SOURCE_DIR}/
00007  get_filename_component(a_dir "${current-shader-path}" PATH)
00008  get_filename_component(a_last_dir "${current-shader-path}" NAME)
00009
00010 set(current-output-path ${a_dir}/spv/${a_last_dir}.spv)
00011 # message(STATUS "${current-output-path}")
00012
00013 get_filename_component(current-output-dir ${current-output-path} DIRECTORY)
00014 file(MAKE_DIRECTORY ${current-output-dir})
00015
00016 add_custom_command(
00017     OUTPUT ${current-output-path}
00018     COMMAND ${GLSLC} --target-env=vulkan1.3 -o ${current-output-path} ${current-shader-path}
00019     DEPENDS ${current-shader-path}
00020     IMPLICIT_DEPENDS CXX ${current-shader-path}
00021     VERBATIM)
00022
00023 # Make sure our build depends on this output.
00024 set_source_files_properties(${current-output-path} PROPERTIES GENERATED TRUE)
00025 target_sources(${TARGET} PRIVATE ${current-output-path})
00026 endfunction(add_shader)

```

## 6.5 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/cmake/Doxygen.cmake File Reference

### 6.6 Doxygen.cmake

[Go to the documentation of this file.](#)

```
00001 function(enable_doxygen)
00002
00003     # first we can indicate the documentation build as an option and set it to ON by default
00004     option(BUILD_DOC "Build documentation" OFF)
00005
00006     # check if Doxygen is installed
00007     if(BUILD_DOC)
00008         find_package(Doxygen)
00009         if (DOXYGEN_FOUND)
00010             # set input and output files
00011             set(DOXYGEN_IN ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in)
00012             set(DOXYGEN_OUT ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile)      # request to configure the file
00013             configure_file(${DOXYGEN_IN} ${DOXYGEN_OUT} @ONLY)
00014             message("Doxygen build started")    # note the option ALL which allows to build the docs
together with the application
00015             add_custom_target( doc_doxygen ALL
00016                 COMMAND ${DOXYGEN_EXECUTABLE} ${DOXYGEN_OUT}
00017                 WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
00018                 COMMENT "Generating API documentation with Doxygen"
00019                 VERBATIM )
00020             else (DOXYGEN_FOUND)
00021                 message("Doxygen need to be installed to generate the doxygen documentation")
00022             endif (DOXYGEN_FOUND)
00023     endif()
00024
00025 endfunction()
```

## 6.7 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetExternalLibsFilters.cmake File Reference

### 6.8 SetExternalLibsFilters.cmake

[Go to the documentation of this file.](#)

```
00001 # setting all project filters
00002 # ---- EXTERNAL LIBS FILTER --- BEGIN
00003
00004 # ---- GUI FILTER --- BEGIN
00005 set(EXTERNAL_LIB_GUI_SRC_DIR ${EXTERNAL_LIB_SRC_DIR}IMGUI/)
00006 set(IMGUI_FILTER ${IMGUI_FILTER})
00007 ${EXTERNAL_LIB_GUI_SRC_DIR}imconfig.h
00008 ${EXTERNAL_LIB_GUI_SRC_DIR}imgui.cpp
00009 ${EXTERNAL_LIB_GUI_SRC_DIR}imgui.h
00010 ${EXTERNAL_LIB_GUI_SRC_DIR}imgui_demo.cpp
00011 ${EXTERNAL_LIB_GUI_SRC_DIR}imgui_draw.cpp
00012 ${EXTERNAL_LIB_GUI_SRC_DIR}imgui_internal.h
00013 ${EXTERNAL_LIB_GUI_SRC_DIR}imgui_tables.cpp
00014 ${EXTERNAL_LIB_GUI_SRC_DIR}imgui_widgets.cpp
00015 ${EXTERNAL_LIB_GUI_SRC_DIR}imstb_rectpack.h
00016 ${EXTERNAL_LIB_GUI_SRC_DIR}imstb_textedit.h
00017 ${EXTERNAL_LIB_GUI_SRC_DIR}imstb_truetype.h
00018 ${EXTERNAL_LIB_GUI_SRC_DIR}backends/imgui_Impl_glfw.h
00019 ${EXTERNAL_LIB_GUI_SRC_DIR}backends/imgui_Impl_glfw.cpp
00020 ${EXTERNAL_LIB_GUI_SRC_DIR}backends/imgui_Impl_vulkan.h
00021 ${EXTERNAL_LIB_GUI_SRC_DIR}backends/imgui_Impl_vulkan.cpp
00022 )
00023 # ---- GUI FILTER --- END
```

## 6.9 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/cmake/filters/SetProjectFilters.cmake File Reference

### 6.10 SetProjectFilters.cmake

[Go to the documentation of this file.](#)

```

00001 # setting all project filters
00002 # ---- PROJECT FILTER --- BEGIN
00003 # ---- GUI FILTER --- BEGIN
00004 set (PROJECT_GUI_SRC_DIR ${PROJECT_SRC_DIR}gui/)
00005 set (PROJECT_GUI_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}gui/)
00006 set (GUI_FILTER ${GUI_FILTER}
00007     ${PROJECT_GUI_SRC_DIR}GUI.cpp
00008     ${PROJECT_GUI_INCLUDE_DIR}GUI.hpp
00009 )
00010 # ---- GUI FILTER --- END
00011
00012 # ---- RENDERER FILTER --- BEGIN
00013 set (PROJECT_RENDERER_SRC_DIR ${PROJECT_SRC_DIR}renderer/)
00014 set (PROJECT_RENDERER_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}renderer/)
00015 set (RENDERER_FILTER ${RENDERER_FILTER}
00016     ${PROJECT_RENDERER_SRC_DIR}VulkanRenderer.cpp
00017     ${PROJECT_RENDERER_INCLUDE_DIR}VulkanRenderer.hpp
00018     ${PROJECT_RENDERER_SRC_DIR}PathTracing.cpp
00019     ${PROJECT_RENDERER_INCLUDE_DIR}PathTracing.hpp
00020     ${PROJECT_RENDERER_SRC_DIR}Raytracing.cpp
00021     ${PROJECT_RENDERER_INCLUDE_DIR}Raytracing.hpp
00022     ${PROJECT_RENDERER_SRC_DIR}Rasterizer.cpp
00023     ${PROJECT_RENDERER_INCLUDE_DIR}Rasterizer.hpp
00024     ${PROJECT_RENDERER_SRC_DIR}PostStage.cpp
00025     ${PROJECT_RENDERER_INCLUDE_DIR}PostStage.hpp
00026     ${PROJECT_RENDERER_SRC_DIR}CommandBufferManager.cpp
00027     ${PROJECT_RENDERER_INCLUDE_DIR}CommandBufferManager.hpp
00028     ${PROJECT_RENDERER_INCLUDE_DIR}GlobalUBO.hpp
00029     ${PROJECT_RENDERER_INCLUDE_DIR}GUIRendererSharedVars.hpp
00030     ${PROJECT_RENDERER_INCLUDE_DIR}QueueFamilyIndices.hpp
00031     ${PROJECT_RENDERER_INCLUDE_DIR}SceneUBO.hpp
00032     ${PROJECT_RENDERER_INCLUDE_DIR}SwapChainDetails.hpp
00033 )
00034 # ---- RENDERER FILTER --- END
00035
00036 # ---- PC FILTER --- BEGIN
00037 set (PROJECT_PC_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}renderer/pushConstants/)
00038 set (PC_FILTER ${PC_FILTER}
00039     ${PROJECT_PC_INCLUDE_DIR}PushConstantPathTracing.hpp
00040     ${PROJECT_PC_INCLUDE_DIR}PushConstantPost.hpp
00041     ${PROJECT_PC_INCLUDE_DIR}PushConstantRasterizer.hpp
00042     ${PROJECT_PC_INCLUDE_DIR}PushConstantRayTracing.hpp
00043 )
00044 # ---- PC FILTER --- END
00045
00046 # ---- AS FILTER --- BEGIN
00047 set (PROJECT_AS_SRC_DIR ${PROJECT_SRC_DIR}renderer/accelerationStructures/)
00048 set (PROJECT_AS_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}renderer/accelerationStructures/)
00049 set (AS_FILTER ${AS_FILTER}
00050     ${PROJECT_AS_INCLUDE_DIR}ASManager.hpp
00051     ${PROJECT_AS_INCLUDE_DIR}BottomLevelAccelerationStructure.hpp
00052     ${PROJECT_AS_INCLUDE_DIR}TopLevelAccelerationStructure.hpp
00053     ${PROJECT_AS_SRC_DIR}ASManager.cpp
00054 )
00055 # ---- AS FILTER --- END
00056
00057 # ---- VULKAN_BASE FILTER --- BEGIN
00058 set (PROJECT_VULKAN_BASE_SRC_DIR ${PROJECT_SRC_DIR}vulkan_base/)
00059 set (PROJECT_VULKAN_BASE_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}vulkan_base/)
00060 set (VULKAN_BASE_FILTER ${VULKAN_BASE_FILTER}
00061     ${PROJECT_VULKAN_BASE_SRC_DIR}ShaderHelper.cpp
00062     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanBuffer.cpp
00063     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanBufferManager.cpp
00064     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanDebug.cpp
00065     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanDevice.cpp
00066     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanImage.cpp
00067     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanImageView.cpp
00068     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanInstance.cpp
00069     ${PROJECT_VULKAN_BASE_SRC_DIR}VulkanSwapChain.cpp
00070
00071     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}ShaderHelper.hpp
00072     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanBuffer.hpp
00073     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanBufferManager.hpp
00074     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanDebug.hpp
00075     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanDevice.hpp

```

```

00076     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanImage.hpp
00077     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanImageView.hpp
00078     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanInstance.hpp
00079     ${PROJECT_VULKAN_BASE_INCLUDE_DIR}VulkanSwapChain.hpp
00080 )
00081 # ---- VULKAN_BASE FILTER --- END
00082
00083 # ---- SCENE FILTER --- BEGIN
00084 set(PROJECT_SCENE_SRC_DIR ${PROJECT_SRC_DIR}scene/)
00085 set(PROJECT_SCENE_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}scene/)
00086 set(SCENE_FILTER ${SCENE_FILTER})
00087     ${PROJECT_SCENE_SRC_DIR}ObjLoader.cpp
00088     ${PROJECT_SCENE_SRC_DIR}Model.cpp
00089     ${PROJECT_SCENE_SRC_DIR}Mesh.cpp
00090     ${PROJECT_SCENE_SRC_DIR}Scene.cpp
00091     ${PROJECT_SCENE_SRC_DIR}Camera.cpp
00092     ${PROJECT_SCENE_SRC_DIR}SceneConfig.cpp
00093     ${PROJECT_SCENE_SRC_DIR}Texture.cpp
00094     ${PROJECT_SCENE_SRC_DIR}Vertex.cpp
00095
00096     ${PROJECT_SCENE_INCLUDE_DIR}ObjMaterial.hpp
00097     ${PROJECT_SCENE_INCLUDE_DIR}Model.hpp
00098     ${PROJECT_SCENE_INCLUDE_DIR}ObjLoader.hpp
00099     ${PROJECT_SCENE_INCLUDE_DIR}Mesh.hpp
00100    ${PROJECT_SCENE_INCLUDE_DIR}Vertex.hpp
00101    ${PROJECT_SCENE_INCLUDE_DIR}Scene.hpp
00102    ${PROJECT_SCENE_INCLUDE_DIR}SceneConfig.hpp
00103    ${PROJECT_SCENE_INCLUDE_DIR}GUISceneSharedVars.hpp
00104    ${PROJECT_SCENE_INCLUDE_DIR}ObjectDescription.hpp
00105    ${PROJECT_SCENE_INCLUDE_DIR}Texture.hpp
00106    ${PROJECT_SCENE_INCLUDE_DIR}Camera.hpp
00107
00108 )
00109 # ---- SCENE FILTER --- END
00110
00111 # ---- WINDOW FILTER --- BEGIN
00112 set(PROJECT_WINDOW_SRC_DIR ${PROJECT_SRC_DIR}window/)
00113 set(PROJECT_WINDOW_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}window/)
00114 set(WINDOW_FILTER ${WINDOW_FILTER})
00115     ${PROJECT_WINDOW_SRC_DIR}Window.cpp
00116     ${PROJECT_WINDOW_INCLUDE_DIR}Window.hpp
00117 )
00118 # ---- WINDOW FILTER --- END
00119
00120 # ---- UTIL FILTER --- BEGIN
00121 set(PROJECT_UTIL_SRC_DIR ${PROJECT_SRC_DIR}util/)
00122 set(PROJECT_UTIL_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}util/)
00123 set(UTIL_FILTER ${UTIL_FILTER})
00124     ${PROJECT_UTIL_SRC_DIR}File.cpp
00125     ${PROJECT_UTIL_INCLUDE_DIR}File.hpp
00126 )
00127 # ---- UTIL FILTER --- END
00128
00129 # ---- APP FILTER --- BEGIN
00130 set(PROJECT_APP_SRC_DIR ${PROJECT_SRC_DIR}app/)
00131 set(PROJECT_APP_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}app/)
00132 set(APP_FILTER ${APP_FILTER})
00133     ${PROJECT_APP_SRC_DIR}App.cpp
00134     ${PROJECT_APP_INCLUDE_DIR}App.hpp
00135 )
00136 # ---- APP FILTER --- END
00137
00138 # ---- COMMON FILTER --- BEGIN
00139 set(PROJECT_COMMON_SRC_DIR ${PROJECT_SRC_DIR}common/)
00140 set(PROJECT_COMMON_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}common/)
00141 set(COMMON_FILTER ${COMMON_FILTER})
00142     ${PROJECT_COMMON_INCLUDE_DIR}FormatHelper.hpp
00143     ${PROJECT_COMMON_INCLUDE_DIR}Globals.hpp
00144     ${PROJECT_COMMON_INCLUDE_DIR}MemoryHelper.hpp
00145     ${PROJECT_COMMON_INCLUDE_DIR}Utilities.hpp
00146 )
00147 # ---- COMMON FILTER --- END
00148
00149 # ---- MEMORY FILTER --- BEGIN
00150 set(PROJECT_MEMORY_SRC_DIR ${PROJECT_SRC_DIR}memory/)
00151 set(PROJECT_MEMORY_INCLUDE_DIR ${PROJECT_INCLUDE_DIR}memory/)
00152 set(MEMORY_FILTER ${MEMORY_FILTER})
00153     ${PROJECT_MEMORY_INCLUDE_DIR}Allocator.hpp
00154     ${PROJECT_MEMORY_SRC_DIR}Allocator.cpp
00155 )
00156 # ---- MEMORY FILTER --- END
00157
00158 # ---- MAIN FILTER --- BEGIN
00159 set(PROJECT_MAIN_SRC_DIR ${PROJECT_SRC_DIR})
00160 set(MAIN_FILTER ${MAIN_FILTER})
00161     ${PROJECT_MAIN_SRC_DIR}Main.cpp
00162 )

```

```
00163 # ---- MAIN FILTER --- END
```

## 6.11 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine Vulkan/cmake/filters/SetShaderFilters.cmake File Reference

## 6.12 SetShaderFilters.cmake

[Go to the documentation of this file.](#)

```
00001 # setting all shader filters
00002 # ---- SHADER FILTER --- BEGIN
00003
00004 # ---- SHADER RASTERIZER FILTER --- BEGIN
00005 set(SHADER_RASTERIZER_SRC_DIR ${SHADER_SRC_DIR}rasterizer/)
00006 set(RASTER_SHADER_FILTER ${RASTER_SHADER_FILTER})
00007     ${SHADER_RASTERIZER_SRC_DIR}shader.vert
00008     ${SHADER_RASTERIZER_SRC_DIR}shader.frag
00009 )
00010 # ---- SHADER RASTERIZER FILTER --- END
00011
00012 # ---- SHADER RAYTRACING FILTER --- BEGIN
00013 set(SHADER_RAYTRACING_SRC_DIR ${SHADER_SRC_DIR}raytracing/)
00014 set(RAYTRACING_SHADER_FILTER ${RAYTRACING_SHADER_FILTER})
00015     ${SHADER_RAYTRACING_SRC_DIR}raytrace.rchit
00016     ${SHADER_RAYTRACING_SRC_DIR}raytrace.rgen
00017     ${SHADER_RAYTRACING_SRC_DIR}raytrace.rmiss
00018     ${SHADER_RAYTRACING_SRC_DIR}shadow.rmiss
00019 )
00020 # ---- SHADER RASTERIZER FILTER --- END
00021
00022 # ---- SHADER COMMON FILTER --- BEGIN
00023 set(SHADER_COMMON_SRC_DIR ${SHADER_SRC_DIR}common/)
00024 set(COMMON_SHADER_FILTER ${COMMON_SHADER_FILTER})
00025     ${SHADER_COMMON_SRC_DIR}Matlib.gsls
00026     ${SHADER_COMMON_SRC_DIR}microfacet.gsls
00027     ${SHADER_COMMON_SRC_DIR}raycommon.gsls
00028     ${SHADER_COMMON_SRC_DIR}ShadingLibrary.gsls
00029 )
00030 # ---- SHADER COMMON FILTER --- END
00031
00032 # ---- SHADER POST FILTER --- BEGIN
00033 set(SHADER_POST_SRC_DIR ${SHADER_SRC_DIR}post/)
00034 set(POST_SHADER_FILTER ${POST_SHADER_FILTER})
00035     ${SHADER_POST_SRC_DIR}post.vert
00036     ${SHADER_POST_SRC_DIR}post.frag
00037 )
00038 # ---- SHADER POST FILTER --- END
00039
00040 # ---- SHADER PATH_TRACING FILTER --- BEGIN
00041 set(SHADER_PATH_TRACING_SRC_DIR ${SHADER_SRC_DIR}path_tracing/)
00042 set(PATH_TRACING_SHADER_FILTER ${PATH_TRACING_SHADER_FILTER})
00043     ${SHADER_PATH_TRACING_SRC_DIR}path_tracing.comp
00044 )
00045 # ---- SHADER PATH_TRACING FILTER --- END
00046
00047 # ---- SHADER BRDF FILTER --- BEGIN
00048 set(SHADER_BRDF_SRC_DIR ${SHADER_SRC_DIR}brdf/)
00049 set(BRDF_SHADER_FILTER ${BRDF_SHADER_FILTER})
00050     ${SHADER_BRDF_SRC_DIR}disney.gsls
00051     ${SHADER_BRDF_SRC_DIR}frostbite.gsls
00052     ${SHADER_BRDF_SRC_DIR}pbrBook.gsls
00053     ${SHADER_BRDF_SRC_DIR}phong.gsls
00054     ${SHADER_BRDF_SRC_DIR}unreal4.gsls
00055 )
00056 # ---- SHADER BRDF FILTER --- END
00057
00058 # ---- SHADER HOST_DEVICE FILTER --- BEGIN
00059 set(SHADER_HOST_DEVICE_SRC_DIR ${SHADER_SRC_DIR}hostDevice/)
00060 set(SHADER_HOST_DEVICE_FILTER ${SHADER_HOST_DEVICE_FILTER})
00061     ${SHADER_HOST_DEVICE_SRC_DIR}host_device_shared_vars.hpp
00062 )
00063 # ---- SHADER HOST_DEVICE FILTER --- END
00064
00065 # ---- SHADER FILTER --- END
```

## 6.13 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/← Sanitizers.cmake File Reference

### 6.14 Sanitizers.cmake

[Go to the documentation of this file.](#)

```
00001 function(enable_sanitizers project_name)
00002
00003     if(CMAKE_CXX_COMPILER_ID STREQUAL "GNU" OR CMAKE_CXX_COMPILER_ID STREQUAL "Clang")
00004         option(ENABLE_COVERAGE "Enable coverage reporting for gcc/clang" OFF)
00005
00006     if(ENABLE_COVERAGE)
00007         target_compile_options(${project_name} INTERFACE --coverage -g)
00008         target_link_libraries(${project_name} INTERFACE --coverage)
00009     endif()
00010
00011     option(ENABLE_SANITIZER_ADDRESS "Enable address sanitizer for gcc/clang" OFF)
00012     set(SANITIZERS "")
00013     if(ENABLE_SANITIZER_ADDRESS)
00014         list(APPEND SANITIZERS "address")
00015     endif()
00016
00017     option(ENABLE_SANITIZER_UNDEFINED_BEHAVIOUR "Enable undefined behaviour sanitizer for
00018     gcc/clang" OFF)
00019     if(ENABLE_SANITIZER_UNDEFINED_BEHAVIOUR)
00020         list(APPEND SANITIZERS "undefined")
00021     endif()
00022
00023     option(ENABLE_SANITIZER_THREAD "Enable thread sanitizer for gcc/clang" OFF)
00024     if(ENABLE_SANITIZER_THREAD)
00025         list(APPEND SANITIZERS "thread")
00026     endif()
00027
00028     list(JOIN SANITIZERS "," LIST_OF_SANITIZERS)
00029     endif()
00030
00031     if(NOT "${LIST_OF_SANITIZERS}" STREQUAL "")
00032         target_compile_options(${project_name} INTERFACE -fsanitize=${LIST_OF_SANITIZERS})
00033         target_link_libraries(${project_name} INTERFACE -fsanitize=${LIST_OF_SANITIZERS})
00034     endif()
00035     endif()
00036
00037 endfunction()
```

## 6.15 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/Set← SourceGroups.cmake File Reference

### 6.16 SetSourceGroups.cmake

[Go to the documentation of this file.](#)

```
00001 # hint for the IDE which belongs together
00002 source_group("vulkan_base"
00003 source_group("gui"
00004 source_group("gui/imgui"
00005 source_group("renderer"
00006 source_group("renderer/pc"
00007 source_group("renderer/AS"
00008 source_group("scene"
00009 source_group("window"
00010 source_group("memory"
00011 source_group("common"
00012 source_group("app"
00013 source_group("util"
00014
00015 source_group("shaders/hostDevice/" FILES ${SHADER_HOST_DEVICE_FILTER})
00016 source_group("shaders/rasterizer/" FILES ${RASTER_SHADER_FILTER})
00017 source_group("shaders/raytracing/" FILES ${RAYTRACING_SHADER_FILTER})
00018 source_group("shaders/common/" FILES ${COMMON_SHADER_FILTER})
00019 source_group("shaders/post/" FILES ${POST_SHADER_FILTER})
00020 source_group("shaders/brdf/" FILES ${BRDF_SHADER_FILTER})
00021 source_group("shaders/path_tracing/" FILES ${PATH_TRACING_SHADER_FILTER})
```

## 6.17 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/StaticAnalyzers.cmake File Reference

### 6.18 StaticAnalyzers.cmake

[Go to the documentation of this file.](#)

```
00001 option(ENABLE_CPPCHECK "Enable cppcheck" OFF) #ON
00002 if(ENABLE_CPPCHECK)
00003     find_program(CPPCHECK cppcheck)
00004     if(CPPCHECK)
00005         set(CMAKE_CXX_CPPCHECK ${CPPCHECK} --suppress=missingInclude --enable=all
00006             --suppressions-list=Documents/cppcheck/suppressions.txt)
00007     endif(CPPCHECK)
00008 endif()
00009 option(ENABLE_CLANGTIDY "Enable clangtidy" OFF) #ON
00010 if(ENABLE_CLANGTIDY)
00011     find_program(CLANGTIDY clang-tidy)
00012     if(CLANGTIDY)
00013         set(CMAKE_CXX_CLANG_TIDY ${CLANGTIDY})
00014     endif(CLANGTIDY)
00015 endif()
```

## 6.19 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/app/App.hpp File Reference

### Data Structures

- class [App](#)

### 6.20 App.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 class App {
00003 public:
00004     App();
00005     int run();
00006     ~App();
00007 private:
00008 };
00009
00010 };
```

## 6.21 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/common/FormatHelper.hpp File Reference

### Functions

- static VkFormat [choose\\_supported\\_format](#) (VkPhysicalDevice physical\_device, const std::vector< VkFormat > &formats, VkImageTiling tiling, VkFormatFeatureFlags feature\_flags)

## 6.21.1 Function Documentation

### 6.21.1.1 choose\_supported\_format()

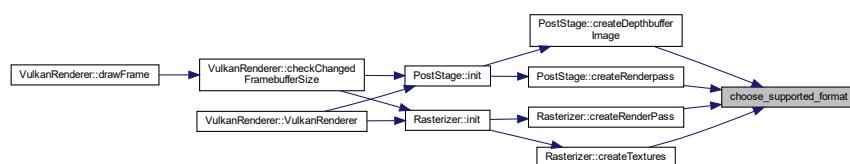
```
static VkFormat choose_supported_format (
    VkPhysicalDevice physical_device,
    const std::vector< VkFormat > & formats,
    VkImageTiling tiling,
    VkFormatFeatureFlags feature_flags ) [static]
```

Definition at line 8 of file FormatHelper.hpp.

```
00011
00012 // loop through options and find compatible one
00013 for (VkFormat format : formats) {
00014     // get properties for give format on this device
00015     VkFormatProperties properties;
00016     vkGetPhysicalDeviceFormatProperties(physical_device, format, &properties);
00017
00018     // depending on tiling choice, need to check for different bit flag
00019     if (tiling == VK_IMAGE_TILING_LINEAR &&
00020         (properties.linearTilingFeatures & feature_flags) == feature_flags) {
00021         return format;
00022     } else if (tiling == VK_IMAGE_TILING_OPTIMAL &&
00023             (properties.optimalTilingFeatures & feature_flags) ==
00024                 feature_flags) {
00025         return format;
00026     }
00027 }
00028 }
00029 throw std::runtime_error("Failed to find supported format!");
00030 }
```

Referenced by [PostStage::createDepthbufferImage\(\)](#), [PostStage::createRenderpass\(\)](#), [Rasterizer::createRenderPass\(\)](#), and [Rasterizer::createTextures\(\)](#).

Here is the caller graph for this function:



## 6.22 FormatHelper.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002
00003 #include <vulkan/vulkan.h>
00004
00005 #include <stdexcept>
00006 #include <vector>
00007
00008 static VkFormat choose_supported_format(VkPhysicalDevice physical_device,
00009                                         const std::vector<VkFormat>& formats,
00010                                         VkImageTiling tiling,
00011                                         VkFormatFeatureFlags feature_flags) {
00012     // loop through options and find compatible one
```

```

00013     for (VkFormat format : formats) {
00014         // get properties for give format on this device
00015         VkFormatProperties properties;
00016         vkGetPhysicalDeviceFormatProperties(physical_device, format, &properties);
00017
00018         // depending on tiling choice, need to check for different bit flag
00019         if (tiling == VK_IMAGE_TILING_LINEAR &&
00020             (properties.linearTilingFeatures & feature_flags) == feature_flags) {
00021             return format;
00022
00023         } else if (tiling == VK_IMAGE_TILING_OPTIMAL &&
00024             (properties.optimalTilingFeatures & feature_flags) ==
00025                 feature_flags) {
00026             return format;
00027         }
00028     }
00029
00030     throw std::runtime_error("Failed to find supported format!");
00031 }
```

## 6.23 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/common/Globals.hpp File Reference

### Variables

- const int [MAX\\_FRAME\\_DRAWNS](#) = 3
- const int [MAX\\_OBJECTS](#) = 40

#### 6.23.1 Variable Documentation

##### 6.23.1.1 MAX\_FRAME\_DRAWNS

```
const int MAX_FRAME_DRAWNS = 3
```

Definition at line 2 of file [Globals.hpp](#).

Referenced by [VulkanRenderer::cleanUpSync\(\)](#), [GUI::create\\_gui\\_context\(\)](#), [VulkanRenderer::createSynchronization\(\)](#), and [VulkanRenderer::drawFrame\(\)](#).

##### 6.23.1.2 MAX\_OBJECTS

```
const int MAX_OBJECTS = 40
```

Definition at line 3 of file [Globals.hpp](#).

Referenced by [VulkanRenderer::createDescriptorPoolSharedRenderStages\(\)](#).

## 6.24 Globals.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 const int MAX_FRAME_DRAWNS = 3;
00003 const int MAX_OBJECTS = 40;
```

## 6.25 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine Vulkan/include/common/MemoryHelper.hpp File Reference

### Functions

- static uint32\_t [align\\_up](#) (uint32\_t memory, uint32\_t alignment)
- static uint32\_t [find\\_memory\\_type\\_index](#) (VkPhysicalDevice physical\_device, uint32\_t allowed\_types, VkMemoryPropertyFlags properties)

#### 6.25.1 Function Documentation

##### 6.25.1.1 align\_up()

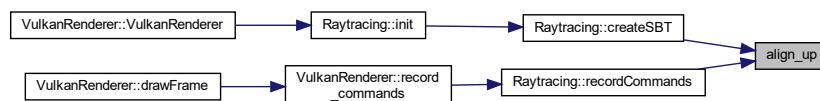
```
static uint32_t align_up (
    uint32_t memory,
    uint32_t alignment ) [static]
```

Definition at line 5 of file [MemoryHelper.hpp](#).

```
00005
00006     return (memory + alignment - 1) & ~(alignment - 1);
00007 }
```

Referenced by [Raytracing::createSBT\(\)](#), and [Raytracing::recordCommands\(\)](#).

Here is the caller graph for this function:



##### 6.25.1.2 find\_memory\_type\_index()

```
static uint32_t find_memory_type_index (
    VkPhysicalDevice physical_device,
    uint32_t allowed_types,
    VkMemoryPropertyFlags properties ) [static]
```

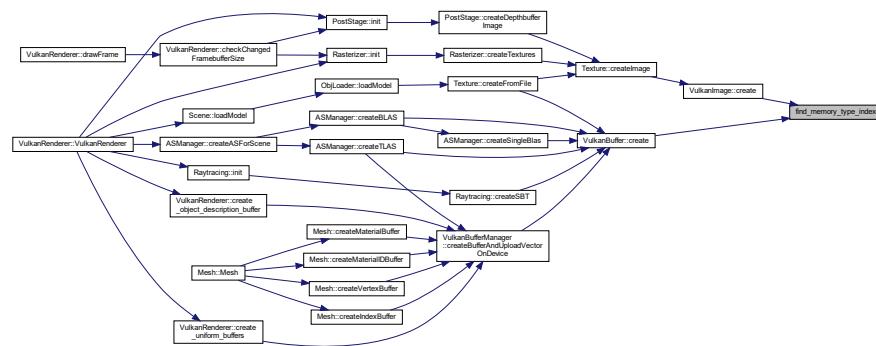
Definition at line 9 of file [MemoryHelper.hpp](#).

```
00011
00012 // get properties of physical device memory
00013 VkPhysicalDeviceMemoryProperties memory_properties{};
00014 vkGetPhysicalDeviceMemoryProperties(physical_device, &memory_properties);
00015 for (uint32_t i = 0; i < memory_properties.memoryTypeCount; i++) {
00016     if ((allowed_types & (1 << i))
00017         // index of memory type must match corresponding bit in allowedTypes
```

```
00019     // desired property bit flags are part of memory type's property flags
00020     && (memory_properties.memoryTypes[i].propertyFlags & properties) ==
00021         properties) {
00022     // this memory type is valid, so return its index
00023     return i;
00024 }
00025 }
00026
00027 return -1;
00028 }
```

Referenced by [VulkanImage::create\(\)](#), and [VulkanBuffer::create\(\)](#).

Here is the caller graph for this function:



## 6.26 MemoryHelper.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 // aligned piece of memory appropriately and when necessary return bigger piece
00005 static uint32_t align_up(uint32_t memory, uint32_t alignment) {
00006     return (memory + alignment - 1) & ~(alignment - 1);
00007 }
00008
00009 static uint32_t find_memory_type_index(VkPhysicalDevice physical_device,
00010                                         uint32_t allowed_types,
00011                                         VkMemoryPropertyFlags properties) {
00012     // get properties of physical device memory
00013     VkPhysicalDeviceMemoryProperties memory_properties{};
00014     vkGetPhysicalDeviceMemoryProperties(physical_device, &memory_properties);
00015
00016     for (uint32_t i = 0; i < memory_properties.memoryTypeCount; i++) {
00017         if ((allowed_types & (1 << i))
00018             // index of memory type must match corresponding bit in allowedTypes
00019             // desired property bit flags are part of memory type's property flags
00020             && (memory_properties.memoryTypes[i].propertyFlags & properties) ==
00021                 properties) {
00022             // this memory type is valid, so return its index
00023             return i;
00024     }
00025 }
00026
00027 return -1;
00028 }
```

## 6.27 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/common/Utilities.hpp File Reference

## Variables

- const bool ENABLE\_VALIDATION\_LAYERS = false

### 6.27.1 Variable Documentation

#### 6.27.1.1 ENABLE\_VALIDATION\_LAYERS

```
const bool ENABLE_VALIDATION_LAYERS = false
```

Definition at line 16 of file [Utilities.hpp](#).

Referenced by [VulkanInstance::VulkanInstance\(\)](#), and [VulkanRenderer::VulkanRenderer\(\)](#).

## 6.28 Utilities.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002
00003 #include <stdexcept>
00004
00005 #include "host_device_shared_vars.hpp"
00006
00007 // Error checking on vulkan function calls
00008 #define ASSERT_VULKAN(val, error_string) \
00009     if (val != VK_SUCCESS) { \
00010         throw std::runtime_error(error_string); \
00011     }
00012
00013 #define NOT_YET_IMPLEMENTED throw std::runtime_error("Not yet implemented!");
00014
00015 #ifdef NDEBUG
00016 const bool ENABLE_VALIDATION_LAYERS = false;
00017 #else
00018 const bool ENABLE_VALIDATION_LAYERS = true;
00019 #endif
```

## 6.29 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/gui/← GUI.hpp File Reference

### Data Structures

- class [GUI](#)

## 6.30 GUI.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002
00003 #include <memory>
00004 #define GLFW_INCLUDE_NONE
00005 #define GLFW_INCLUDE_VULKAN
00006 #include <GLFW/glfw3.h>
00007 #include <imgui.h>
00008 #include <imgui_impl_glfw.h>
00009 #include <imgui_impl_vulkan.h>
00010
00011 #include "CommandBufferManager.hpp"
00012 #include "GUIRendererSharedVars.hpp"
00013 #include "GUISceneSharedVars.hpp"
```

```

00014 #include "Globals.hpp"
00015 #include "VulkanDevice.hpp"
00016 #include "Window.hpp"
00017
00018 class GUI {
00019 public:
00020     GUI(Window* window);
00021
00022     void initializeVulkanContext(VulkanDevice* device, const VkInstance& instance,
00023                                     const VkRenderPass& post_render_pass,
00024                                     const VkCommandPool& graphics_command_pool);
00025
00026     GUISceneSharedVars getGuiSceneSharedVars() { return guiSceneSharedVars; };
00027     GUIRendererSharedVars& getGuiRendererSharedVars() {
00028         return guiRendererSharedVars;
00029     };
00030
00031     void render();
00032
00033     void cleanUp();
00034
00035     ~GUI();
00036
00037 private:
00038     void create_gui_context(Window* window, const VkInstance& instance,
00039                             const VkRenderPass& post_render_pass);
00040
00041     void create_fonts_and_upload(const VkCommandPool& graphics_command_pool);
00042
00043     VulkanDevice* device{VK_NULL_HANDLE};
00044     Window* window{VK_NULL_HANDLE};
00045     VkDescriptorPool gui_descriptor_pool{VK_NULL_HANDLE};
00046     CommandBufferManager commandBufferManager;
00047
00048     GUISceneSharedVars guiSceneSharedVars;
00049     GUIRendererSharedVars guiRendererSharedVars;
00050 };

```

## 6.31 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ← Vulkan/include/memory/Allocator.hpp File Reference

### Data Structures

- class Allocator

## 6.32 Allocator.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <vk_mem_alloc.h>
00003 #include <vulkan/vulkan.h>
00004
00005 #include <stdexcept>
00006
00007 class Allocator {
00008 public:
00009     Allocator();
0010     Allocator(const VkDevice& device, const VkPhysicalDevice& physicalDevice,
0011               const VkInstance& instance);
0012
0013     void cleanUp();
0014
0015     ~Allocator();
0016
0017 private:
0018     VmaAllocator vmaAllocator;
0019 };

```

## 6.33 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/include/renderer/accelerationStructures/ASManager.hpp File Reference

### Data Structures

- struct [BuildAccelerationStructure](#)
- struct [BlasInput](#)
- class [ASManager](#)

## 6.34 ASManager.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include "BottomLevelAccelerationStructure.hpp"
00005 #include "CommandBufferManager.hpp"
00006 #include "Scene.hpp"
00007 #include "TopLevelAccelerationStructure.hpp"
00008 #include "VulkanDevice.hpp"
00009
00010 struct BuildAccelerationStructure {
00011     VkAccelerationStructureBuildGeometryInfoKHR build_info;
00012     VkAccelerationStructureBuildSizesInfoKHR size_info;
00013     const VkAccelerationStructureBuildRangeInfoKHR* range_info;
00014     BottomLevelAccelerationStructure single_bias;
00015 };
00016
00017 struct BlasInput {
00018     std::vector<VkAccelerationStructureGeometryKHR> as_geometry;
00019     std::vector<VkAccelerationStructureBuildRangeInfoKHR> as_build_offset_info;
00020 };
00021
00022 class ASManager {
00023 public:
00024     ASManager();
00025
00026     VkAccelerationStructureKHR& getTLAS() { return tlas.vulkanAS; }
00027
00028     void createASForScene(VulkanDevice* device, VkCommandPool commandPool,
00029                           Scene* scene);
00030
00031     void createBLAS(VulkanDevice* device, VkCommandPool commandPool,
00032                      Scene* scene);
00033
00034     void createTLAS(VulkanDevice* device, VkCommandPool commandPool,
00035                      Scene* scene);
00036
00037     void cleanUp();
00038
00039     ~ASManager();
00040
00041 private:
00042     VulkanDevice* vulkanDevice{VK_NULL_HANDLE};
00043     CommandBufferManager commandBufferManager;
00044     VulkanBufferManager vulkanBufferManager;
00045
00046     std::vector<BottomLevelAccelerationStructure> blas;
00047     TopLevelAccelerationStructure tlas;
00048
00049     void createSingleBlas(VulkanDevice* device, VkCommandBuffer command_buffer,
00050                           BuildAccelerationStructure& build_as_structure,
00051                           VkDeviceAddress scratch_device_or_host_address);
00052
00053     void createAccelerationStructureInfosBLAS(
00054         VulkanDevice* device, BuildAccelerationStructure& build_as_structure,
00055         BlasInput& blas_input, VkDeviceSize& current_scratch_size,
00056         VkDeviceSize& current_size);
00057
00058     void objectToVkGeometryKHR(
00059         VulkanDevice* device, Mesh* mesh,
00060         VkAccelerationStructureGeometryKHR& acceleration_structure_geometry,
00061         VkAccelerationStructureBuildRangeInfoKHR&
00062             acceleration_structure_build_range_info);
00063 };
```

## 6.35 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngine/Vulkan/include/renderer/accelerationStructures/BottomLevel/AccelerationStructure.hpp File Reference

### Data Structures

- struct [BottomLevelAccelerationStructure](#)

## 6.36 BottomLevelAccelerationStructure.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include "VulkanBuffer.hpp"
00005
00006 struct BottomLevelAccelerationStructure {
00007     VkAccelerationStructureKHR vulkanAS;
00008     VulkanBuffer vulkanBuffer;
00009 };
```

## 6.37 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngine/Vulkan/include/renderer/accelerationStructures/TopLevel/AccelerationStructure.hpp File Reference

### Data Structures

- struct [TopLevelAccelerationStructure](#)

## 6.38 TopLevelAccelerationStructure.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002
00003 #include <vulkan/vulkan.h>
00004
00005 #include "VulkanBuffer.hpp"
00006
00007 struct TopLevelAccelerationStructure {
00008     VkAccelerationStructureKHR vulkanAS;
00009     VulkanBuffer vulkanBuffer;
00010 };
```

## 6.39 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngine/Vulkan/include/renderer/CommandBufferManager.hpp File Reference

### Data Structures

- class [CommandBufferManager](#)

## 6.40 CommandBufferManager.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include "Utilities.hpp"
00005
00006 class CommandBufferManager {
00007 public:
00008     CommandBufferManager();
00009
0010     VkCommandBuffer beginCommandBuffer(VkDevice device,
0011                                         VkCommandPool command_pool);
0012     void endAndSubmitCommandBuffer(VkDevice device, VkCommandPool command_pool,
0013                                    VkQueue queue,
0014                                    VkCommandBuffer& command_buffer);
0015
0016     ~CommandBufferManager();
0017
0018 private:
0019 };
```

## 6.41 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/GlobalUBO.hpp File Reference

### Data Structures

- struct [GlobalUBO](#)

### Typedefs

- using `vec2` = `glm::vec2`
- using `vec3` = `glm::vec3`
- using `vec4` = `glm::vec4`
- using `mat4` = `glm::mat4`
- using `uint` = `unsigned int`

### 6.41.1 Typedef Documentation

#### 6.41.1.1 mat4

```
using mat4 = glm::mat4
```

Definition at line 13 of file [GlobalUBO.hpp](#).

#### 6.41.1.2 uint

```
using uint = unsigned int
```

Definition at line 14 of file [GlobalUBO.hpp](#).

### 6.41.1.3 vec2

```
using vec2 = glm::vec2
```

Definition at line 10 of file [GlobalUBO.hpp](#).

### 6.41.1.4 vec3

```
using vec3 = glm::vec3
```

Definition at line 11 of file [GlobalUBO.hpp](#).

### 6.41.1.5 vec4

```
using vec4 = glm::vec4
```

Definition at line 12 of file [GlobalUBO.hpp](#).

## 6.42 GlobalUBO.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :)
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <glm/glm.hpp>
00009 // GLSL Type
00010 using vec2 = glm::vec2;
00011 using vec3 = glm::vec3;
00012 using vec4 = glm::vec4;
00013 using mat4 = glm::mat4;
00014 using uint = unsigned int;
00015 #endif
00016
00017 // which render stage doesn't need view,projection ?
00018 struct GlobalUBO {
00019     mat4 projection;
00020     mat4 view;
00021 };
```

## 6.43 C:/Users/jonas\_16e3q/Desktop/GraphicsEngine Vulkan/include/renderer/GUIRendererSharedVars.hpp File Reference

### Data Structures

- struct [GUIRendererSharedVars](#)

## 6.44 GUIRendererSharedVars.hpp

[Go to the documentation of this file.](#)

```
00001 struct GUIRendererSharedVars {
00002     bool raytracing = false;
00003     bool pathTracing = false;
00004
00005     bool shader_hot_reload_triggered = false;
00006
00007     // path tracing vars
00008 };
```

## 6.45 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine Vulkan/include/renderer/PathTracing.hpp File Reference

### Data Structures

- class [PathTracing](#)
- struct [PathTracing::SpecializationData](#)

## 6.46 PathTracing.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002
00003 #include <vulkan/vulkan.h>
00004
00005 #include "PushConstantPathTracing.hpp"
00006 #include "VulkanDevice.hpp"
00007 #include "VulkanSwapChain.hpp"
00008
00009 class PathTracing {
00010     public:
00011     PathTracing();
00012
00013     void init(VulkanDevice* device,
00014                 const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts);
00015
00016     void shaderHotReload(
00017         const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts);
00018
00019     void recordCommands(VkCommandBuffer& commandBuffer, uint32_t image_index,
00020                         VulkanImage& vulkanImage,
00021                         VulkanSwapChain* vulkanSwapChain,
00022                         const std::vector<VkDescriptorSet>& descriptorSets);
00023
00024     void cleanUp();
00025
00026     ~PathTracing();
00027
00028 private:
00029     VulkanDevice* device{VK_NULL_HANDLE};
00030
00031     VkPipelineLayout pipeline_layout{VK_NULL_HANDLE};
00032     VKPipeline pipeline{VK_NULL_HANDLE};
00033     VkPushConstantRange pc_range{VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0, 0};
00034     PushConstantPathTracing push_constant{glm::vec4(0.f), 0, 0};
00035
00036     float timeStampPeriod{0};
00037     uint64_t pathTracingTiming{static_cast<uint64_t>(-1.f)};
00038     uint32_t query_count{2};
00039     std::vector<uint64_t> queryResults;
00040     VkQueryPool queryPool{VK_NULL_HANDLE};
00041
00042     struct {
00043         uint32_t maxComputeWorkGroupCount[3] = {static_cast<uint32_t>(-1),
00044                                                 static_cast<uint32_t>(-1),
00045                                                 static_cast<uint32_t>(-1)};
00046         uint32_t maxComputeWorkGroupInvocations = -1;
00047         uint32_t maxComputeWorkGroupSize[3] = {static_cast<uint32_t>(-1),
```

```

00048                         static_cast<uint32_t>(-1),
00049                         static_cast<uint32_t>(-1)};
00050
00051     } computeLimits;
00052
00053     struct SpecializationData {
00054         // standard values
00055         uint32_t specWorkGroupSizeX = 16;
00056         uint32_t specWorkGroupSizeY = 8;
00057         uint32_t specWorkGroupSizeZ = 0;
00058     };
00059
00060     SpecializationData specializationData;
00061
00062     void createQueryPool();
00063     void createPipeline(
00064         const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts);
00065 };

```

## 6.47 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ← Vulkan/include/renderer/PostStage.hpp File Reference

### Data Structures

- class PostStage

## 6.48 PostStage.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include "PushConstantPost.hpp"
00003 #include "VulkanDevice.hpp"
00004 #include "VulkanSwapChain.hpp"
00005
00006 #include <vulkan/vulkan.h>
00007
00008 class PostStage {
00009 public:
0010     PostStage();
0011
0012     void init(VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
0013               const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts);
0014
0015     void shaderHotReload(
0016         const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts);
0017
0018     VkRenderPass& getRenderPass() { return render_pass; };
0019     VkSampler& getOffscreenSampler() { return offscreenTextureSampler; };
0020
0021     void recordCommands(VkCommandBuffer& commandBuffer, uint32_t image_index,
0022                          const std::vector<VkDescriptorSet>& descriptorSets);
0023
0024     void cleanUp();
0025
0026     ~PostStage();
0027
0028 private:
0029     VulkanDevice* device{VK_NULL_HANDLE};
0030     VulkanSwapChain* vulkanSwapChain{VK_NULL_HANDLE};
0031
0032     std::vector<VkFramebuffer> framebuffers;
0033     Texture depthBufferImage;
0034     VkFormat depth_format{VK_FORMAT_UNDEFINED};
0035     void createDepthbufferImage();
0036
0037     VkSampler offscreenTextureSampler;
0038     void createOffscreenTextureSampler();
0039
0040     VkPushConstantRange push_constant_range{VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0,
0041                                             0};
0041     VkRenderPass render_pass{VK_NULL_HANDLE};
0042     VkPipeline graphics_pipeline{VK_NULL_HANDLE};
0043     VkPipelineLayout pipeline_layout{VK_NULL_HANDLE};
0044

```

```
00045 void createPushConstantRange();
00046 void createRenderpass();
00047 void createGraphicsPipeline(
00048     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts);
00049 void createFramebuffer();
00050 };
```

## 6.49 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/pushConstants/PushConstantPathTracing.hpp File Reference

### Data Structures

- struct [PushConstantPathTracing](#)

### Typedefs

- using `vec2` = `glm::vec2`
- using `vec3` = `glm::vec3`
- using `vec4` = `glm::vec4`
- using `mat4` = `glm::mat4`
- using `uint` = `unsigned int`

#### 6.49.1 Typedef Documentation

##### 6.49.1.1 mat4

```
using mat4 = glm::mat4
```

Definition at line 13 of file [PushConstantPathTracing.hpp](#).

##### 6.49.1.2 uint

```
using uint = unsigned int
```

Definition at line 14 of file [PushConstantPathTracing.hpp](#).

##### 6.49.1.3 vec2

```
using vec2 = glm::vec2
```

Definition at line 10 of file [PushConstantPathTracing.hpp](#).

#### 6.49.1.4 vec3

```
using vec3 = glm::vec3
```

Definition at line 11 of file [PushConstantPathTracing.hpp](#).

#### 6.49.1.5 vec4

```
using vec4 = glm::vec4
```

Definition at line 12 of file [PushConstantPathTracing.hpp](#).

## 6.50 PushConstantPathTracing.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :)
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <glm/glm.hpp>
00009 // GLSL Type
00010 using vec2 = glm::vec2;
00011 using vec3 = glm::vec3;
00012 using vec4 = glm::vec4;
00013 using mat4 = glm::mat4;
00014 using uint = unsigned int;
00015 #endif
00016
00017 struct PushConstantPathTracing {
00018     vec4 clearColor;
00019     uint width;
00020     uint height;
00021 };
```

## 6.51 C:/Users/jonas\_16e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/pushConstants/PushConstantPost.hpp File Reference

### Data Structures

- struct [PushConstantPost](#)

### Typedefs

- using **vec2** = glm::vec2
- using **vec3** = glm::vec3
- using **vec4** = glm::vec4
- using **mat4** = glm::mat4
- using **uint** = unsigned int

## 6.51.1 Typedef Documentation

### 6.51.1.1 mat4

```
using mat4 = glm::mat4
```

Definition at line 13 of file [PushConstantPost.hpp](#).

### 6.51.1.2 uint

```
using uint = unsigned int
```

Definition at line 14 of file [PushConstantPost.hpp](#).

### 6.51.1.3 vec2

```
using vec2 = glm::vec2
```

Definition at line 10 of file [PushConstantPost.hpp](#).

### 6.51.1.4 vec3

```
using vec3 = glm::vec3
```

Definition at line 11 of file [PushConstantPost.hpp](#).

### 6.51.1.5 vec4

```
using vec4 = glm::vec4
```

Definition at line 12 of file [PushConstantPost.hpp](#).

## 6.52 PushConstantPost.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :(
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <glm/glm.hpp>
00009 // GLSL Type
00010 using vec2 = glm::vec2;
00011 using vec3 = glm::vec3;
00012 using vec4 = glm::vec4;
00013 using mat4 = glm::mat4;
00014 using uint = unsigned int;
00015 #endif
00016
00017 struct PushConstantPost {
00018     float aspect_ratio;
00019 };
```

## 6.53 C:/Users/jonas\_16e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/pushConstants/PushConstant ↵ Rasterizer.hpp File Reference

### Data Structures

- struct [PushConstantRasterizer](#)

### Typedefs

- using [vec2](#) = glm::vec2
- using [vec3](#) = glm::vec3
- using [vec4](#) = glm::vec4
- using [mat4](#) = glm::mat4
- using [uint](#) = unsigned int

### 6.53.1 Typedef Documentation

#### 6.53.1.1 mat4

```
using mat4 = glm::mat4
```

Definition at line 13 of file [PushConstantRasterizer.hpp](#).

### 6.53.1.2 uint

```
using uint = unsigned int
```

Definition at line 14 of file [PushConstantRasterizer.hpp](#).

### 6.53.1.3 vec2

```
using vec2 = glm::vec2
```

Definition at line 10 of file [PushConstantRasterizer.hpp](#).

### 6.53.1.4 vec3

```
using vec3 = glm::vec3
```

Definition at line 11 of file [PushConstantRasterizer.hpp](#).

### 6.53.1.5 vec4

```
using vec4 = glm::vec4
```

Definition at line 12 of file [PushConstantRasterizer.hpp](#).

## 6.54 PushConstantRasterizer.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :)
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <glm/glm.hpp>
00009 // GLSL Type
00010 using vec2 = glm::vec2;
00011 using vec3 = glm::vec3;
00012 using vec4 = glm::vec4;
00013 using mat4 = glm::mat4;
00014 using uint = unsigned int;
00015 #endif
00016
00017 // Push constant structure for the raster
00018 struct PushConstantRasterizer {
00019     mat4 model; // matrix of the instance
00020 };
```

## 6.55 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/pushConstants/PushConstantRay ↵ Tracing.hpp File Reference

### Data Structures

- struct [PushConstantRaytracing](#)

### Typedefs

- using `vec2` = `glm::vec2`
- using `vec3` = `glm::vec3`
- using `vec4` = `glm::vec4`
- using `mat4` = `glm::mat4`
- using `uint` = `unsigned int`

#### 6.55.1 Typedef Documentation

##### 6.55.1.1 mat4

```
using mat4 = glm::mat4
```

Definition at line 13 of file [PushConstantRayTracing.hpp](#).

##### 6.55.1.2 uint

```
using uint = unsigned int
```

Definition at line 14 of file [PushConstantRayTracing.hpp](#).

##### 6.55.1.3 vec2

```
using vec2 = glm::vec2
```

Definition at line 10 of file [PushConstantRayTracing.hpp](#).

### 6.55.1.4 vec3

```
using vec3 = glm::vec3
```

Definition at line 11 of file [PushConstantRayTracing.hpp](#).

### 6.55.1.5 vec4

```
using vec4 = glm::vec4
```

Definition at line 12 of file [PushConstantRayTracing.hpp](#).

## 6.56 PushConstantRayTracing.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :)
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <glm/glm.hpp>
00009 // GLSL Type
0010 using vec2 = glm::vec2;
0011 using vec3 = glm::vec3;
0012 using vec4 = glm::vec4;
0013 using mat4 = glm::mat4;
0014 using uint = unsigned int;
0015 #endif
0016
0017 struct PushConstantRaytracing {
0018     vec4 clear_color;
0019 };
```

## 6.57 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine Vulkan/include/renderer/QueueFamilyIndices.hpp File Reference

### Data Structures

- struct [QueueFamilyIndices](#)

## 6.58 QueueFamilyIndices.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 // Indices (locations) of Queue families (if they exist at all)
00003 struct QueueFamilyIndices {
00004     int graphics_family = -1;           // location of graphics family
00005     int presentation_family = -1;        // location of presentation queue family
00006     int compute_family = -1;            // location of compute queue family
00007
00008     // check if queue families are valid
00009     bool is_valid() {
0010         return graphics_family >= 0 && presentation_family >= 0 &&
0011             compute_family >= 0;
0012     }
0013 };
```

## 6.59 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/Rasterizer.hpp File Reference

### Data Structures

- class [Rasterizer](#)

## 6.60 Rasterizer.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include "PushConstantRasterizer.hpp"
00005 #include "Scene.hpp"
00006 #include "Texture.hpp"
00007 #include "VulkanDevice.hpp"
00008 #include "VulkanSwapChain.hpp"
00009
00010 class Rasterizer {
00011 public:
00012     Rasterizer();
00013
00014     void init(VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
00015                 const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts,
00016                 VkCommandPool& commandPool);
00017
00018     void shaderHotReload(
00019         const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts);
00020
00021     Texture& getOffscreenTexture(uint32_t index);
00022
00023     void setPushConstant(PushConstantRasterizer pushConstant);
00024
00025     void recordCommands(VkCommandBuffer& commandBuffer, uint32_t image_index,
00026                          Scene* scene,
00027                          const std::vector<VkDescriptorSet>& descriptorSets);
00028
00029     void cleanUp();
00030
00031     ~Rasterizer();
00032
00033 private:
00034     VulkanDevice* device{VK_NULL_HANDLE};
00035     VulkanSwapChain* vulkanSwapChain{VK_NULL_HANDLE};
00036
00037     CommandBufferManager commandBufferManager;
00038
00039     std::vector<VkFramebuffer> framebuffer;
00040     std::vector<Texture> offscreenTextures;
00041     Texture depthBufferImage;
00042
00043     VkPushConstantRange push_constant_range{VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0,
00044                                         0};
00045     PushConstantRasterizer pushConstant{glm::mat4(1.f)};
00046
00047     VkPipeline graphics_pipeline{VK_NULL_HANDLE};
00048     VkPipelineLayout pipeline_layout{VK_NULL_HANDLE};
00049     VkRenderPass render_pass{VK_NULL_HANDLE};
00050
00051     void createTextures(VkCommandPool& commandPool);
00052     void createGraphicsPipeline(
00053         const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts);
00054     void createRenderPass();
00055     void createFramebuffer();
00056     void createPushConstantRange();
00057 };

```

## 6.61 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/Raytracing.hpp File Reference

### Data Structures

- class [Raytracing](#)

## 6.62 Raytracing.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002
00003 #include <vulkan/vulkan.h>
00004
00005 #include "PushConstantRayTracing.hpp"
00006 #include "VulkanBuffer.hpp"
00007 #include "VulkanSwapChain.hpp"
00008
00009 class Raytracing {
00010     public:
00011     Raytracing();
00012
00013     void init(VulkanDevice* device,
00014             const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts);
00015
00016     void shaderHotReload(
00017         const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts);
00018
00019     void recordCommands(VkCommandBuffer& commandBuffer,
00020                         VulkanSwapChain* vulkanSwapChain,
00021                         const std::vector<VkDescriptorSet>& descriptorSets);
00022
00023     void cleanUp();
00024
00025     ~Raytracing();
00026
00027     private:
00028     VulkanDevice* device{VK_NULL_HANDLE};
00029     VulkanSwapChain* vulkanSwapChain{VK_NULL_HANDLE};
00030
00031     VkPipeline graphicsPipeline{VK_NULL_HANDLE};
00032     VkPipelineLayout pipeline_layout{VK_NULL_HANDLE};
00033     PushConstantRaytracing pc{glm::vec4(0.f)};
00034     VkPushConstantRange pc_ranges{VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM, 0, 0};
00035
00036     std::vector<VkRayTracingShaderGroupCreateInfoKHR> shader_groups;
00037     VulkanBuffer shaderBindingTableBuffer;
00038     VulkanBuffer raygenShaderBindingTableBuffer;
00039     VulkanBuffer missShaderBindingTableBuffer;
00040     VulkanBuffer hitShaderBindingTableBuffer;
00041
00042     VkStridedDeviceAddressRegionKHR rgen_region{};
00043     VkStridedDeviceAddressRegionKHR miss_region{};
00044     VkStridedDeviceAddressRegionKHR hit_region{};
00045     VkStridedDeviceAddressRegionKHR call_region{};
00046
00047     VkPhysicalDeviceRayTracingPipelinePropertiesKHR raytracing_properties{};
00048
00049     void createPCRange();
00050     void createGraphicsPipeline(
00051         const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts);
00052     void createSBT();
00053 };

```

## 6.63 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/SceneUBO.hpp File Reference

### Data Structures

- struct SceneUBO

### Typedefs

- using `vec2` = `glm::vec2`
- using `vec3` = `glm::vec3`
- using `vec4` = `glm::vec4`
- using `mat4` = `glm::mat4`
- using `uint` = `unsigned int`

### 6.63.1 Typedef Documentation

#### 6.63.1.1 mat4

```
using mat4 = glm::mat4
```

Definition at line 13 of file [SceneUBO.hpp](#).

#### 6.63.1.2 uint

```
using uint = unsigned int
```

Definition at line 14 of file [SceneUBO.hpp](#).

#### 6.63.1.3 vec2

```
using vec2 = glm::vec2
```

Definition at line 10 of file [SceneUBO.hpp](#).

#### 6.63.1.4 vec3

```
using vec3 = glm::vec3
```

Definition at line 11 of file [SceneUBO.hpp](#).

#### 6.63.1.5 vec4

```
using vec4 = glm::vec4
```

Definition at line 12 of file [SceneUBO.hpp](#).

## 6.64 SceneUBO.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :)
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <glm/glm.hpp>
00009 // GLSL Type
00010 using vec2 = glm::vec2;
00011 using vec3 = glm::vec3;
00012 using vec4 = glm::vec4;
00013 using mat4 = glm::mat4;
00014 using uint = unsigned int;
00015 #endif
00016
00017 struct SceneUBO {
00018     vec4 light_dir;
00019     vec4 view_dir;
00020     // xyz is position; w = fov
00021     vec4 cam_pos;
00022};
```

## 6.65 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/SwapChainDetails.hpp File Reference

### Data Structures

- struct [SwapChainDetails](#)

## 6.66 SwapChainDetails.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include <vector>
00005
00006 struct SwapChainDetails {
00007     // surface properties, e.g. image size/extent
00008     VkSurfaceCapabilitiesKHR surface_capabilities;
00009     // surface image formats, e.g. RGBA and size of each color
00010     std::vector<VkSurfaceFormatKHR> formats;
00011     // how images should be presented to screen
00012     std::vector<VkPresentModeKHR> presentation_mode;
00013};
```

## 6.67 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/renderer/VulkanRenderer.hpp File Reference

### Data Structures

- class [VulkanRenderer](#)

## 6.68 VulkanRenderer.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002
00003 #define GLFW_INCLUDE_NONE
00004 #define GLFW_INCLUDE_VULKAN
00005
00006 #include <GLFW/glfw3.h>
00007 #include <stdio.h>
00008 #include <stdlib.h>
00009
00010 #include <algorithm>
00011 #include <array>
00012 #include <cstring>
00013 #include <glm/glm.hpp>
00014 #include <glm/gtc/matrix_transform.hpp>
00015 #include <iostream>
00016 #include <memory>
00017 #include <set>
00018 #include <sstream>
00019 #include <stdexcept>
00020 #include <vector>
00021
00022 #include "ASManager.hpp"
00023 #include "Allocator.hpp"
00024 #include "CommandBufferManager.hpp"
00025 #include "GUI.hpp"
00026 #include "GUISceneSharedVars.hpp"
00027 #include "GlobalUBO.hpp"
00028 #include "PathTracing.hpp"
00029 #include "PostStage.hpp"
00030 #include "PushConstantRasterizer.hpp"
00031 #include "PushConstantRayTracing.hpp"
00032 #include "QueueFamilyIndices.hpp"
00033 #include "Rasterizer.hpp"
00034 #include "Raytracing.hpp"
00035 #include "Scene.hpp"
00036 #include "SceneUBO.hpp"
00037 #include "Texture.hpp"
00038 #include "Utilities.hpp"
00039 #include "VulkanBuffer.hpp"
00040 #include "VulkanBufferManager.hpp"
00041 #include "VulkanDevice.hpp"
00042 #include "VulkanInstance.hpp"
00043 #include "VulkanSwapChain.hpp"
00044 #include "Window.hpp"
00045
00046 class VulkanRenderer {
00047 public:
00048     VulkanRenderer(Window* window, Scene* scene, GUI* gui, Camera* camera);
00049
00050     void drawFrame();
00051
00052     void updateUniforms(Scene* scene, Camera* camera, Window* window);
00053
00054     void updateStateDueToUserInput(GUI* gui);
00055     void finishAllRenderCommands();
00056     void update_raytracing_descriptor_set(uint32_t image_index);
00057
00058     void cleanUp();
00059
00060     ~VulkanRenderer();
00061
00062 private:
00063     void shaderHotReload();
00064
00065     // helper class for managing our buffers
00066     VulkanBufferManager vulkanBufferManager;
00067
00068     // Vulkan instance, stores all per-application states
00069     VulkanInstance instance;
00070
00071     // surface defined on windows as WIN32 window system, Linux f.e. X11, MacOS
00072     // also their own
00073     VkSurfaceKHR surface;
00074     void create_surface();
00075
00076     std::unique_ptr<VulkanDevice> device;
00077
00078     VulkanSwapChain vulkanSwapChain;
00079
00080     Window* window;
00081     Scene* scene;
00082     GUI* gui;

```

```

00083
00084 // -- pools
00085 void record_commands(uint32_t image_index);
00086 void create_command_pool();
00087 void cleanUpCommandPools();
00088 VkCommandPool graphics_command_pool;
00089 VkCommandPool compute_command_pool;
00090
00091 // uniform buffers
00092 GlobalUBO globalUBO;
00093 std::vector<VulkanBuffer> globalUBOBuffer;
00094 SceneUBO sceneUBO;
00095 std::vector<VulkanBuffer> sceneUBOBuffer;
00096 void create_uniform_buffers();
00097 void update_uniform_buffers(uint32_t image_index);
00098 void cleanUpUBOs();
00099
00100 std::vector<VkCommandBuffer> command_buffers;
00101 CommandBufferManager commandBufferManager;
00102 void create_command_buffers();
00103
00104 Raytracing raytracingStage;
00105 Rasterizer rasterizer;
00106 PathTracing pathTracing;
00107 PostStage postStage;
00108
00109 // new era of memory management for my project
00110 // for now on integrate vma
00111 Allocator allocator;
00112
00113 // -- synchronization
00114 uint32_t current_frame{0};
00115 std::vector<VkSemaphore> image_available;
00116 std::vector<VkSemaphore> render_finished;
00117 std::vector<VkFence> in_flight_fences;
00118 std::vector<VkFence> images_in_flight_fences;
00119 void createSynchronization();
00120 void cleanUpSync();
00121
00122 ASManager asManager;
00123 VulkanBuffer objectDescriptionBuffer;
00124 void create_object_description_buffer();
00125
00126 VkDescriptorPool descriptorPoolSharedRenderStages;
00127 void createDescriptorPoolSharedRenderStages();
00128 VkDescriptorSetLayout sharedRenderDescriptorsetLayout;
00129 void createSharedRenderDescriptorsetLayouts();
00130 std::vector<VkDescriptorSet> sharedRenderDescriptorSet;
00131 void createSharedRenderDescriptorSet();
00132 void updateTexturesInSharedRenderDescriptorSet();
00133
00134 VkDescriptorPool post_descriptor_pool(VK_NULL_HANDLE);
00135 VkDescriptorSetLayout post_descriptor_set_layout(VK_NULL_HANDLE);
00136 std::vector<VkDescriptorSet> post_descriptor_set;
00137 void create_post_descriptor_layout();
00138 void updatePostDescriptorSets();
00139
00140 VkDescriptorPool raytracingDescriptorPool(VK_NULL_HANDLE);
00141 std::vector<VkDescriptorSet> raytracingDescriptorSet;
00142 VkDescriptorSetLayout raytracingDescriptorsetLayout(VK_NULL_HANDLE);
00143
00144 void createRaytracingDescriptorsetLayouts();
00145 void createRaytracingDescriptorSets();
00146 void updateRaytracingDescriptorSets();
00147 void createRaytracingDescriptorPool();
00148
00149 bool checkChangedFrameBufferSize();
00150 };

```

## 6.69 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine Vulkan/include/renderer/VulkanRendererConfig.h File Reference

### 6.70 VulkanRendererConfig.h

[Go to the documentation of this file.](#)

```
00001 // the configured options and settings for renderer
```

```

00002 //
https://riptutorial.com/cmake/example/32603/using-cmake-to-define-the-version-number-for-cplusplus-usage
00003 #ifndef RENDERER_CONFIG_GUARD
00004 #define RENDERER_CONFIG_GUARD
00005
00006 #define VulkanRenderer_VERSION_MAJOR "1"
00007 #define VulkanRenderer_VERSION_MINOR "3"
00008
00009 #define VULKAN_VERSION_MAJOR "1"
00010 #define VULKAN_VERSION_MINOR "3"
00011
00012 #define GLSLC_EXE "C:/VulkanSDK/1.3.211.0/Bin/glslc.exe"
00013
00014 #define RELATIVE_RESOURCE_PATH "../Resources/"
00015 #define RELATIVE_IMGUI_FONTS_PATH "../ExternalLib/IMGUI/misc/fonts/"
00016
00017 #endif

```

## 6.71 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine/Vulkan/include/renderer/VulkanRendererConfig.hpp File Reference

### 6.72 VulkanRendererConfig.hpp

[Go to the documentation of this file.](#)

```

00001 // the configured options and settings for renderer
00002 //
https://riptutorial.com/cmake/example/32603/using-cmake-to-define-the-version-number-for-cplusplus-usage
00003 #ifndef RENDERER_CONFIG_GUARD
00004 #define RENDERER_CONFIG_GUARD
00005
00006 #define VulkanRenderer_VERSION_MAJOR "1"
00007 #define VulkanRenderer_VERSION_MINOR "3"
00008
00009 #define VULKAN_VERSION_MAJOR "1"
00010 #define VULKAN_VERSION_MINOR "3"
00011
00012 #define GLSLC_EXE "C:/VulkanSDK/1.3.211.0/Bin/glslc.exe"
00013
00014 #define RELATIVE_RESOURCE_PATH "../Resources/"
00015 #define RELATIVE_IMGUI_FONTS_PATH "../ExternalLib/IMGUI/misc/fonts/"
00016
00017 #endif

```

## 6.73 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine/Vulkan/include/scene/Camera.hpp File Reference

### Data Structures

- class [Camera](#)

### 6.74 Camera.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002
00003 #include <glm/glm.hpp>
00004 #include <glm/gtc/matrix_transform.hpp>
00005
00006 #define GLFW_INCLUDE_NONE
00007 #define GLFW_INCLUDE_VULKAN
00008
00009 #include <GLFW/glfw3.h>

```

```
00010
00011 class Camera {
00012 public:
00013     Camera();
00014
00015     void key_control(bool* keys, float delta_time);
00016     void mouse_control(float x_change, float y_change);
00017
00018     glm::vec3 get_camera_position() const { return position; };
00019     glm::vec3 get_camera_direction() const { return glm::normalize(front); };
00020     glm::vec3 get_up_axis() const { return up; };
00021     glm::vec3 get_right_axis() const { return right; };
00022     float get_near_plane() const { return near_plane; };
00023     float get_far_plane() const { return far_plane; };
00024     float get_fov() const { return fov; };
00025     float get_yaw() const { return yaw; };
00026
00027     glm::mat4 calculate_viewmatrix();
00028
00029     void set_near_plane(float near_plane);
00030     void set_far_plane(float far_plane);
00031     void set_fov(float fov);
00032     void set_camera_position(glm::vec3 new_camera_position);
00033
00034     ~Camera();
00035
00036 private:
00037     glm::vec3 position;
00038     glm::vec3 front;
00039     glm::vec3 world_up;
00040     glm::vec3 right;
00041     glm::vec3 up;
00042
00043     float yaw;
00044     float pitch;
00045
00046     float movement_speed;
00047     float turn_speed;
00048
00049     float near_plane, far_plane, fov;
00050
00051     void update();
00052 };
```

## 6.75 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine← Vulkan/include/scene/GUISceneSharedVars.hpp File Reference

### Data Structures

- struct [GUISceneSharedVars](#)

## 6.76 GUISceneSharedVars.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 struct GUISceneSharedVars {
00003     float direccional_light_radiance = 10.f;
00004     float directional_light_color[3] = {1.f, 1.f, 1.f};
00005     float directional_light_direction[3] = {0.075f, -1.f, 0.118f};
00006 };
```

## 6.77 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine← Vulkan/include/scene/Mesh.hpp File Reference

### Data Structures

- class [Mesh](#)

## 6.78 Mesh.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <glm/glm.hpp>
00003 #include <vector>
00004
00005 #include "ObjMaterial.hpp"
00006 #include "ObjectDescription.hpp"
00007 #include "Utilities.hpp"
00008 #include "Vertex.hpp"
00009 #include "VulkanBufferManager.hpp"
00010
00011 // this a simple Mesh without mesh generation
00012 class Mesh {
00013 public:
00014     Mesh(VulkanDevice* device, VkQueue transfer_queue,
00015           VkCommandPool transfer_command_pool, std::vector<Vertex>& vertices,
00016           std::vector<uint32_t>& indices, std::vector<unsigned int>& materialIndex,
00017           std::vector<ObjMaterial>& materials);
00018
00019     Mesh();
00020
00021     void cleanUp();
00022
00023     ObjectDescription& getObjectDescription() { return object_description; };
00024     glm::mat4 getModel() { return model; };
00025     uint32_t getVertexCount() { return vertex_count; };
00026     uint32_t getIndexCount() { return index_count; };
00027     VulkanBuffer& getVertexBuffer() { return vertexBuffer.getBuffer(); };
00028     VulkanBuffer& getMaterialIDBuffer() { return materialIdsBuffer.getBuffer(); };
00029     VulkanBuffer& getIndexBuffer() { return indexBuffer.getBuffer(); };
00030
00031     void setModel(glm::mat4 new_model);
00032
00033     ~Mesh();
00034
00035 private:
00036     VulkanBufferManager vulkanBufferManager;
00037
00038     ObjectDescription object_description{
00039         static_cast<uint64_t>(-1), static_cast<uint64_t>(-1),
00040         static_cast<uint64_t>(-1), static_cast<uint64_t>(-1)};
00041
00042     VulkanBuffer vertexBuffer;
00043     VulkanBuffer indexBuffer;
00044     VulkanBuffer objectDescriptionBuffer;
00045     VulkanBuffer materialIdsBuffer;
00046     VulkanBuffer materialsBuffer;
00047
00048     glm::mat4 model;
00049
00050     uint32_t vertex_count{static_cast<uint32_t>(-1)};
00051     uint32_t index_count{static_cast<uint32_t>(-1)};
00052
00053     VulkanDevice* device{VK_NULL_HANDLE};
00054
00055     void createVertexBuffer(VkQueue transfer_queue,
00056                             VkCommandPool transfer_command_pool,
00057                             std::vector<Vertex>& vertices);
00058
00059     void createIndexBuffer(VkQueue transfer_queue,
00060                           VkCommandPool transfer_command_pool,
00061                           std::vector<uint32_t>& indices);
00062
00063     void createMaterialIDBuffer(VkQueue transfer_queue,
00064                                 VkCommandPool transfer_command_pool,
00065                                 std::vector<unsigned int>& materialIndex);
00066
00067     void createMaterialBuffer(VkQueue transfer_queue,
00068                              VkCommandPool transfer_command_pool,
00069                              std::vector<ObjMaterial>& materials);
00070 };

```

## 6.79 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/scene/Model.hpp File Reference

### Data Structures

- class [Model](#)

## 6.80 Model.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <iostream>
00003 #include <memory>
00004 #include <unordered_map>
00005 #include <vector>
00006
00007 #include "Mesh.hpp"
00008 #include "Texture.hpp"
00009
00010 class Model {
00011 public:
00012     Model();
00013     Model(VulkanDevice* device);
00014
00015     void cleanUp();
00016
00017     void add_new_mesh(VulkanDevice* device, VkQueue transfer_queue,
00018                         VkCommandPool command_pool, std::vector<Vertex>& vertices,
00019                         std::vector<unsigned int>& indices,
00020                         std::vector<unsigned int>& materialIndex,
00021                         std::vector<ObjMaterial>& materials);
00022
00023     uint32_t getTextureCount() {
00024         return static_cast<uint32_t>(modelTextures.size());
00025     }
00026     std::vector<Texture>& getTextures() { return modelTextures; }
00027     std::vector<VkSampler>& getTextureSamplers() { return modelTextureSamplers; }
00028     std::vector<std::string> getTextureList() { return texture_list; }
00029     uint32_t getMeshCount() { return 1; }
00030     Mesh* getMesh(size_t index) { return &mesh; }
00031     glm::mat4 getModel() { return model; }
00032     uint32_t getCustomInstanceIndex() { return mesh_model_index; }
00033     uint32_t getPrimitiveCount();
00034     ObjectDescription getObjectDescription() {
00035         return mesh.getObjectDescription();
00036     }
00037
00038     void set_model(glm::mat4 model);
00039     void addTexture(Texture newTexture);
00040
00041     ~Model();
00042
00043 private:
00044     VulkanDevice* device{VK_NULL_HANDLE};
00045
00046     void addSampler(Texture newTexture);
00047
00048     uint32_t mesh_model_index{static_cast<uint32_t>(-1)};
00049     Mesh mesh;
00050     glm::mat4 model;
00051
00052     std::vector<std::string> texture_list;
00053     std::vector<Texture> modelTextures;
00054     std::vector<VkSampler> modelTextureSamplers;
00055 };

```

## 6.81 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine Vulkan/include/scene/ObjectDescription.hpp File Reference

### Data Structures

- struct [ObjectDescription](#)

## 6.82 ObjectDescription.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
```

```

00002 // CPU side as well for the GPU side :)
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <vulkan/vulkan.h>
00009 #endif
0010
0011 struct ObjectDescription {
0012     uint64_t vertex_address;
0013     uint64_t index_address;
0014     uint64_t material_index_address;
0015     uint64_t material_address;
0016 };

```

## 6.83 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine/Vulkan/include/scene/ObjLoader.hpp File Reference

### Data Structures

- class [ObjLoader](#)

## 6.84 ObjLoader.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include <memory>
00005
00006 #include "Model.hpp"
00007 #include "ObjMaterial.hpp"
00008 #include "Vertex.hpp"
00009
0010 class ObjLoader {
0011 public:
0012     ObjLoader(VulkanDevice* device, VkQueue transfer_queue,
0013               VkCommandPool command_pool);
0014
0015     std::shared_ptr<Model> loadModel(const std::string& modelFile);
0016
0017 private:
0018     VulkanDevice* device;
0019     VkQueue transfer_queue;
0020     VkCommandPool command_pool;
0021
0022     std::vector<Vertex> vertices;
0023     std::vector<unsigned int> indices;
0024     std::vector<ObjMaterial> materials;
0025     std::vector<unsigned int> materialIndex;
0026     std::vector<std::string> textures;
0027
0028     std::vector<std::string> loadTexturesAndMaterials(
0029         const std::string& modelFile);
0030     void loadVertices(const std::string& fileName);
0031 };

```

## 6.85 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine/Vulkan/include/scene/ObjMaterial.hpp File Reference

### Data Structures

- struct [ObjMaterial](#)

## Typedefs

- using `vec2` = `glm::vec2`
- using `vec3` = `glm::vec3`
- using `vec4` = `glm::vec4`
- using `mat4` = `glm::mat4`
- using `uint` = `unsigned int`

### 6.85.1 Typedef Documentation

#### 6.85.1.1 mat4

```
using mat4 = glm::mat4
```

Definition at line 13 of file [ObjMaterial.hpp](#).

#### 6.85.1.2 uint

```
using uint = unsigned int
```

Definition at line 14 of file [ObjMaterial.hpp](#).

#### 6.85.1.3 vec2

```
using vec2 = glm::vec2
```

Definition at line 10 of file [ObjMaterial.hpp](#).

#### 6.85.1.4 vec3

```
using vec3 = glm::vec3
```

Definition at line 11 of file [ObjMaterial.hpp](#).

#### 6.85.1.5 vec4

```
using vec4 = glm::vec4
```

Definition at line 12 of file [ObjMaterial.hpp](#).

## 6.86 ObjMaterial.hpp

[Go to the documentation of this file.](#)

```
00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :(
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005
00006 #ifdef __cplusplus
00007 #pragma once
00008 #include <glm/glm.hpp>
00009 // GLSL Type
00010 using vec2 = glm::vec2;
00011 using vec3 = glm::vec3;
00012 using vec4 = glm::vec4;
00013 using mat4 = glm::mat4;
00014 using uint = unsigned int;
00015 #endif
00016
00017 // illumination model (see http://www.fileformat.info/format/material/)
00018
00019 struct ObjMaterial {
00020     vec3 ambient;
00021     vec3 diffuse;
00022     vec3 specular;
00023     vec3 transmittance;
00024     vec3 emission;
00025     float shininess;
00026     float ior;           // index of refraction
00027     float dissolve;    // 1 == opaque; 0 == fully transparent
00028
00029     int illum;
00030     int textureID;
00031 };
```

## 6.87 C:/Users/jonas\_16e3q/Desktop/GraphicsEngine ↵ Vulkan/include/scene/Scene.hpp File Reference

### Data Structures

- class [Scene](#)

## 6.88 Scene.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #define GLFW_INCLUDE_NONE
00003 #define GLFW_INCLUDE_VULKAN
00004 #include <GLFW glfw3.h>
00005 #include <stdio.h>
00006 #include <stdlib.h>
00007
00008 #include <algorithm>
00009 #include <array>
00010 #include <cstring>
00011 #include <glm/glm.hpp>
00012 #include <glm/gtc/matrix_transform.hpp>
00013 #include <iostream>
00014 #include <memory>
00015 #include <set>
00016 #include <stdexcept>
00017 #include <string>
00018 #include <vector>
00019
00020 #include "Camera.hpp"
00021 #include "GUI.hpp"
00022 #include "GUISceneSharedVars.hpp"
00023 #include "Mesh.hpp"
00024 #include "Model.hpp"
00025 #include "ObjLoader.hpp"
```

```

00026 #include "SceneConfig.hpp"
00027 #include "Utilities.hpp"
00028 #include "Window.hpp"
00029
00030 class Scene {
00031 public:
00032     Scene();
00033
00034     void update_user_input(GUI* gui);
00035     void update_model_matrix(glm::mat4 model_matrix, int model_id);
00036
00037     const GUISceneSharedVars& getGuiSceneSharedVars() {
00038         return guiSceneSharedVars;
00039     };
00040
00041     std::vector<Texture>& getTextures(int model_index) {
00042         return model_list[model_index]->getTextures();
00043     };
00044     std::vector<VkSampler>& getTextureSampler(int model_index) {
00045         return model_list[model_index]->getTextureSamplers();
00046     };
00047     uint32_t getTextureCount(int model_index) {
00048         return model_list[model_index]->getTextureCount();
00049     };
00050     uint32_t getModelCount() { return static_cast<uint32_t>(model_list.size()); };
00051     glm::mat4 getModelMatrix(int model_index) {
00052         return model_list[model_index]->getModel();
00053     };
00054     uint32_t getMeshCount(int model_index) {
00055         return static_cast<uint32_t>(model_list[model_index]->getMeshCount());
00056     };
00057     VkBuffer getVertexBuffer(int model_index, int mesh_index) {
00058         return model_list[model_index]->getMesh(mesh_index)->getVertexBuffer();
00059     };
00060     VkBuffer getIndexBuffer(int model_index, int mesh_index) {
00061         return model_list[model_index]->getMesh(mesh_index)->getIndexBuffer();
00062     };
00063     uint32_t getIndexCount(int model_index, int mesh_index) {
00064         return model_list[model_index]->getMesh(mesh_index)->getIndexCount();
00065     };
00066     uint32_t getNumberObjectDescriptions() {
00067         return static_cast<uint32_t>(object_descriptions.size());
00068     };
00069     uint32_t getNumberMeshes();
00070     std::vector<ObjectDescription> getObjectDescriptions() {
00071         return object_descriptions;
00072     };
00073     std::vector<std::shared_ptr<Model>> const& get_model_list() {
00074         return model_list;
00075     };
00076
00077     void loadModel(VulkanDevice* device, VkCommandPool commandPool);
00078
00079     void add_model(std::shared_ptr<Model> model);
00080     void add_object_description(ObjectDescription object_description);
00081
00082     void cleanUp();
00083     ~Scene();
00084
00085 private:
00086     std::vector<ObjectDescription> object_descriptions;
00087     std::vector<std::shared_ptr<Model>> model_list;
00088
00089     GUISceneSharedVars guiSceneSharedVars;
00090 };

```

## 6.89 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine Vulkan/include/scene/SceneConfig.hpp File Reference

### Namespaces

- namespace `sceneConfig`

### Functions

- `std::string sceneConfig::getModelFile()`
- `glm::mat4 sceneConfig::getModelMatrix()`

## 6.90 SceneConfig.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <glm/glm.hpp>
00003 #include <glm/gtc/matrix_transform.hpp>
00004 #include <sstream>
00005 #include <string>
00006
00007 namespace sceneConfig {
00008
00009 std::string getModelFile();
00010 glm::mat4 getModelMatrix();
00011
00012 } // namespace sceneConfig
```

## 6.91 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/scene/Texture.hpp File Reference

### Data Structures

- class [Texture](#)

## 6.92 Texture.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <stb_image.h>
00003 #include <vulkan/vulkan.h>
00004
00005 #include <string>
00006
00007 #include "Utilities.hpp"
00008 #include "VulkanBuffer.hpp"
00009 #include "VulkanBufferManager.hpp"
00010 #include "VulkanImage.hpp"
00011 #include "VulkanImageView.hpp"
00012
00013 class Texture {
00014 public:
00015     Texture();
00016
00017     void createFromFile(VulkanDevice* device, VkCommandPool commandPool,
00018                         const std::string& fileName);
00019
00020     void setImage(VkImage image);
00021     void setImageView(VkImageView imageView);
00022
00023     uint32_t getMipLevel() { return mip_levels; };
00024     VulkanImage& getVulkanImage() { return vulkanImage; };
00025     VulkanImageView& getVulkanImageView() { return vulkanImageView; };
00026     VkImage& getImage() { return vulkanImage.getImage(); };
00027     VkImageView& getImageView() { return vulkanImageView.getImageView(); };
00028
00029     void createImage(VulkanDevice* device, uint32_t width, uint32_t height,
00030                      uint32_t mip_levels, VkFormat format, VkImageTiling tiling,
00031                      VkImageUsageFlags use_flags,
00032                      VkMemoryPropertyFlags prop_flags);
00033
00034     void createImageView(VulkanDevice* device, VkFormat format,
00035                          VkImageAspectFlags aspect_flags, uint32_t mip_levels);
00036
00037     void cleanUp();
00038
00039     ~Texture();
00040
00041 private:
00042     uint32_t mip_levels = 0;
00043
00044     stbi_uc* loadTextureData(const std::string& file_name, int* width,
```

```

00045             int* height, VkDeviceSize* image_size);
00046
00047 void generateMipMaps(VkPhysicalDevice physical_device, VkDevice device,
00048                         VkCommandPool command_pool, VkQueue queue, VkImage image,
00049                         VkFormat image_format, int32_t width, int32_t height,
00050                         uint32_t mip_levels);
00051
00052 CommandBufferManager commandBufferManager;
00053 VulkanBufferManager vulkanBufferManager;
00054
00055 VulkanImage vulkanImage;
00056 VulkanImageView vulkanImageView;
00057 };

```

## 6.93 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Vertex.hpp File Reference

### Data Structures

- struct [Vertex](#)
- struct [std::hash< Vertex >](#)

### Namespaces

- namespace [vertex](#)
- namespace [std](#)

### Functions

- std::array< [VkVertexInputAttributeDescription](#), 4 > [vertex::getVertexInputAttributeDesc\(\)](#)

## 6.94 Vertex.hpp

[Go to the documentation of this file.](#)

```

00001 // this little "hack" is needed for using it on the
00002 // CPU side as well for the GPU side :)
00003 // inspired by the NVIDIA tutorial:
00004 // https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
00005 #ifdef __cplusplus
00006 #pragma once
00007 #define GLM_ENABLE_EXPERIMENTAL
00008 #include <vulkan/vulkan.h>
00009
00010 #include <array>
00011 #include <glm/glm.hpp>
00012 #include <glm/gtx/hash.hpp>
00013 #include <vector>
00014
00015 class Vertex {
00016 public:
00017     Vertex();
00018     Vertex(glm::vec3 pos, glm::vec3 normal, glm::vec3 color,
00019             glm::vec2 texture_coords);
00020
00021     glm::vec3 pos;
00022     glm::vec3 normal;
00023     glm::vec3 color;
00024     glm::vec2 texture_coords;
00025
00026     bool operator==(const Vertex& other) const {
00027         return pos == other.pos && normal == other.normal &&
00028                 texture_coords == other.texture_coords;
00029     }

```

```

00030 };
00031
00032 namespace vertex {
00033
00034 std::array<VkVertexInputAttributeDescription, 4> getVertexInputAttributeDesc();
00035
00036 }
00037
00038 namespace std {
00039 template <>
00040 struct hash<Vertex> {
00041     size_t operator()(Vertex const& vertex) const {
00042         size_t h1 = hash<glm::vec3>()(vertex.pos);
00043         size_t h2 = hash<glm::vec3>()(vertex.color);
00044         size_t h3 = hash<glm::vec2>()(vertex.texture_coords);
00045         size_t h4 = hash<glm::vec3>()(vertex.normal);
00046
00047         return (((((h2 << 1) ^ h1) >> 1) ^ h3) << 1) ^ h4));
00048     }
00049 };
00050 } // namespace std
00051 #else
00052 struct Vertex {
00053     vec3 pos;
00054     vec3 normal;
00055     vec3 color;
00056     vec2 texture_coords;
00057 };
00058
00059 #endif

```

## 6.95 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/include/util/← File.hpp File Reference

### Data Structures

- class [File](#)

## 6.96 File.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <string>
00003 #include <vector>
00004
00005 class File {
00006 public:
00007     explicit File(const std::string& file_location);
00008
00009     std::string read();
00010     std::vector<char> readCharSequence();
00011     std::string getBaseDir();
00012
00013     ~File();
00014
00015 private:
00016     std::string file_location;
00017 };

```

## 6.97 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngine← Vulkan/include/vulkan\_base/ShaderHelper.hpp File Reference

### Data Structures

- class [ShaderHelper](#)

## 6.98 ShaderHelper.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include <string>
00005 #include <vector>
00006
00007 #include "VulkanDevice.hpp"
00008
00009 class ShaderHelper {
00010 public:
00011     ShaderHelper();
00012
00013     void compileShader(const std::string& shader_src_dir,
00014                         const std::string& shader_name);
00015     std::string getShaderSpvDir(const std::string& shader_src_dir,
00016                                 const std::string& shader_name);
00017
00018     VkShaderModule createShaderModule(VulkanDevice* device,
00019                                       const std::vector<char>& code);
00020
00021     ~ShaderHelper();
00022
00023 private:
00024     std::string target = " --target-env=vulkan1.3 ";
00025 };
```

## 6.99 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ↵ Vulkan/include/vulkan\_base/VulkanBuffer.hpp File Reference

### Data Structures

- class [VulkanBuffer](#)

## 6.100 VulkanBuffer.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include "VulkanDevice.hpp"
00005
00006 class VulkanBuffer {
00007 public:
00008     VulkanBuffer();
00009
00010     void create(VulkanDevice* vulkanDevice, VkDeviceSize buffer_size,
00011                 VkBufferUsageFlags buffer_usage_flags,
00012                 VkMemoryPropertyFlags buffer_property_flags);
00013
00014     void cleanUp();
00015
00016     VkBuffer& getBuffer() { return buffer; };
00017     VkDeviceMemory& getBufferMemory() { return bufferMemory; };
00018
00019     ~VulkanBuffer();
00020
00021 private:
00022     VulkanDevice* device{VK_NULL_HANDLE};
00023
00024     VkBuffer buffer{VK_NULL_HANDLE};
00025     VkDeviceMemory bufferMemory{VK_NULL_HANDLE};
00026
00027     bool created{false};
00028 };
```

## 6.101 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ← Vulkan/include/vulkan\_base/VulkanBufferManager.hpp File Reference

### Data Structures

- class [VulkanBufferManager](#)

## 6.102 VulkanBufferManager.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include <cstring>
00005 #include <vector>
00006
00007 #include "CommandBufferManager.hpp"
00008 #include "Utilities.hpp"
00009 #include "VulkanBuffer.hpp"
00010
00011 class VulkanBufferManager {
00012 public:
00013     VulkanBufferManager();
00014
00015     void copyBuffer(VkDevice device, VkQueue transfer_queue,
00016                     VKCommandPool transfer_command_pool, VulkanBuffer src_buffer,
00017                     VulkanBuffer dst_buffer, VkDeviceSize buffer_size);
00018
00019     void copyImageBuffer(VkDevice device, VkQueue transfer_queue,
00020                          VKCommandPool transfer_command_pool, VkBuffer src_buffer,
00021                          VkImage image, uint32_t width, uint32_t height);
00022
00023     template <typename T>
00024     void createBufferAndUploadVectorOnDevice(
00025         VulkanDevice* device, VkCommandPool commandPool,
00026         VulkanBuffer& vulkanBuffer, VkBufferUsageFlags dstBufferUsageFlags,
00027         VkMemoryPropertyFlags dstBufferMemoryPropertyFlags, std::vector<T>& data);
00028
00029     ~VulkanBufferManager();
00030
00031 private:
00032     CommandBufferManager commandBufferManager;
00033 };
00034
00035 template <typename T>
00036 inline void VulkanBufferManager::createBufferAndUploadVectorOnDevice(
00037     VulkanDevice* device, VkCommandPool commandPool, VulkanBuffer& vulkanBuffer,
00038     VkBufferUsageFlags dstBufferUsageFlags,
00039     VkMemoryPropertyFlags dstBufferMemoryPropertyFlags,
00040     std::vector<T>& bufferData) {
00041     VkDeviceSize bufferSize = sizeof(T) * bufferData.size();
00042
00043     // temporary buffer to "stage" vertex data before transferring to GPU
00044     VulkanBuffer stagingBuffer;
00045
00046     // create buffer and allocate memory to it
00047     stagingBuffer.create(device, bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
00048                           VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
00049                           VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);
00050
00051     // Map memory to vertex buffer
00052     // 1.) create pointer to a point in normal memory
00053     void* data;
00054     // 2.) map the vertex buffer memory to that point
00055     vkMapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory(), 0,
00056                 bufferSize, 0, &data);
00057     // 3.) copy memory from vertices vector to the point
00058     std::memcpy(data, bufferData.data(), static_cast<size_t>(bufferSize));
00059     // 4.) unmap the vertex buffer memory
00060     vkUnmapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory());
00061
00062     // create buffer with TRANSFER_DST_BIT to mark as recipient of transfer data
00063     // (also VERTEX_BUFFER) buffer memory is to be DEVICE_LOCAL_BIT meaning memory
00064     // is on the GPU and only accessible by it and not CPU (host)

```

```
00065     vulkanBuffer.create(device, bufferSize, dstBufferUsageFlags,
00066                             dstBufferMemoryPropertyFlags);
00067
00068     // copy staging buffer to vertex buffer on GPU
00069     copyBuffer(device->getLogicalDevice(), device->getGraphicsQueue(),
00070                 commandPool, stagingBuffer, vulkanBuffer, bufferSize);
00071
00072     stagingBuffer.cleanUp();
00073 }
```

## 6.103 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/VulkanDebug.hpp File Reference

### Namespaces

- namespace `debug`

### Functions

- `VKAPI_ATTR VkBool32 VKAPI_CALL debug::messageCallback (VkDebugReportFlagsEXT flags, VkDebugReportObjectTypeEXT objType, uint64_t srcObject, size_t location, int32_t msgCode, const char *pLayerPrefix, const char *pMsg, void *pUserData)`
- `void debug::setupDebugging (VkInstance instance, VkDebugReportFlagsEXT flags, VkDebugReportCallbackEXT callBack)`
- `void debug::freeDebugCallback (VkInstance instance)`

### Variables

- `int debug::validationLayerCount`
- `const char * debug::validationLayerNames []`

## 6.104 VulkanDebug.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include <iostream>
00005 #include <sstream>
00006 #include <string>
00007
00008 #include "Utilities.hpp"
00009
00010 namespace debug {
00011     // Default validation layers
00012     extern int validationLayerCount;
00013     extern const char* validationLayerNames[];
00014
00015     // Default debug callback
00016     VKAPI_ATTR VkBool32 VKAPI_CALL
00017     messageCallback(VkDebugReportFlagsEXT flags, VkDebugReportObjectTypeEXT objType,
00018                      uint64_t srcObject, size_t location, int32_t msgCode,
00019                      const char* pLayerPrefix, const char* pMsg, void* pUserData);
00020
00021     // Load debug function pointers and set debug callback
00022     // if callBack is NULL, default message callback will be used
00023     void setupDebugging(VkInstance instance, VkDebugReportFlagsEXT flags,
00024                          VkDebugReportCallbackEXT callBack);
00025     // Clear debug callback
00026     void freeDebugCallback(VkInstance instance);
00027
00028 } // namespace debug
```

## 6.105 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ← Vulkan/include/vulkan\_base/VulkanDevice.hpp File Reference

### Data Structures

- class [VulkanDevice](#)

## 6.106 VulkanDevice.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include <vector>
00005
00006 #include "QueueFamilyIndices.hpp"
00007 #include "SwapChainDetails.hpp"
00008 #include "VulkanInstance.hpp"
00009
00010 class VulkanDevice {
00011 public:
00012     VulkanDevice(VulkanInstance* instance, VkSurfaceKHR* surface);
00013
00014     VkPhysicalDeviceProperties getPhysicalDeviceProperties() {
00015         return device_properties;
00016     };
00017     VkPhysicalDevice getPhysicalDevice() const { return physical_device; };
00018     VkDevice getLogicalDevice() const { return logical_device; };
00019     QueueFamilyIndices getQueueFamilies();
00020     VkQueue getGraphicsQueue() const { return graphics_queue; };
00021     VkQueue getComputeQueue() const { return compute_queue; };
00022     VkQueue getPresentationQueue() const { return presentation_queue; };
00023     SwapChainDetails getSwapchainDetails();
00024
00025     void cleanUp();
00026
00027     ~VulkanDevice();
00028
00029 private:
00030     VKPhysicalDevice physical_device;
00031     VkPhysicalDeviceProperties device_properties;
00032
00033     VkDevice logical_device;
00034
00035     VulkanInstance* instance;
00036     VkSurfaceKHR* surface;
00037
00038     // available queues
00039     VkQueue graphics_queue;
00040     VkQueue presentation_queue;
00041     VkQueue compute_queue;
00042
00043     void get_physical_device();
00044     void create_logical_device();
00045
00046     QueueFamilyIndices getQueueFamilies(VkPhysicalDevice physical_device);
00047     SwapChainDetails getSwapchainDetails(VkPhysicalDevice device);
00048
00049     bool check_device_suitable(VkPhysicalDevice device);
00050     bool check_device_extension_support(VkPhysicalDevice device);
00051
00052     const std::vector<const char*> device_extensions = {
00053
00054         VK_KHR_SWAPCHAIN_EXTENSION_NAME
00055
00056     };
00057
00058     // DEVICE EXTENSIONS FOR RAYTRACING
00059     const std::vector<const char*> device_extensions_for_raytracing = {
00060
00061         // raytracing related extensions
00062         VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME,
00063         VK_KHR_RAY_TRACING_PIPELINE_EXTENSION_NAME,
00064         // required from VK_KHR_acceleration_structure
00065         VK_KHR_BUFFER_DEVICE_ADDRESS_EXTENSION_NAME,
00066         VK_KHR_DEFERRED_HOST_OPERATIONS_EXTENSION_NAME,

```

```
00067     VK_EXT_DESCRIPTOR_INDEXING_EXTENSION_NAME,  
00068     // required for pipeline  
00069     VK_KHR_SPIRV_1_4_EXTENSION_NAME,  
00070     // required by VK_KHR_spirv_1_4  
00071     VK_KHR_SHADER_FLOAT_CONTROLS_EXTENSION_NAME,  
00072     // required for pipeline library  
00073     VK_KHR_PIPELINE_LIBRARY_EXTENSION_NAME,  
00074     // lets start ray queries  
00075     VK_KHR_RAY_QUERY_EXTENSION_NAME  
00076  
00077 };  
00078 };
```

## 6.107 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/VulkanImage.hpp File Reference

### Data Structures

- class [VulkanImage](#)

## 6.108 VulkanImage.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once  
00002 #include <vulkan/vulkan.h>  
00003  
00004 #include "CommandBufferManager.hpp"  
00005 #include "VulkanDevice.hpp"  
00006  
00007 class VulkanImage {  
00008 public:  
00009     VulkanImage();  
00010  
00011     void create(VulkanDevice* device, uint32_t width, uint32_t height,  
00012                 uint32_t mip_levels, VkFormat format, VkImageTiling tiling,  
00013                 VkImageUsageFlags use_flags, VkMemoryPropertyFlags prop_flags);  
00014  
00015     void transitionImageLayout(VkDevice device, VkQueue queue,  
00016                             VkCommandPool command_pool,  
00017                             VkImageLayout old_layout, VkImageLayout new_layout,  
00018                             VkImageAspectFlags aspectMask,  
00019                             uint32_t mip_levels);  
00020  
00021     void transitionImageLayout(VkCommandBuffer command_buffer,  
00022                             VkImageLayout old_layout, VkImageLayout new_layout,  
00023                             uint32_t mip_levels,  
00024                             VkImageAspectFlags aspectMask);  
00025  
00026     void setImage(VkImage image);  
00027     VkImage& getImage() { return image; }  
00028  
00029     void cleanUp();  
00030  
00031     ~VulkanImage();  
00032  
00033 private:  
00034     VulkanDevice* device{VK_NULL_HANDLE};  
00035     CommandBufferManager commandBufferManager;  
00036  
00037     VkImage image;  
00038     VkDeviceMemory imageMemory;  
00039  
00040     VkAccessFlags accessFlagsForImageLayout(VkImageLayout layout);  
00041     VkPipelineStageFlags pipelineStageForLayout(VkImageLayout oldImageLayout);  
00042 };
```

## 6.109 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/include/vulkan\_base/VulkanImageView.hpp File Reference

### Data Structures

- class [VulkanImageView](#)

## 6.110 VulkanImageView.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003
00004 #include "VulkanDevice.hpp"
00005
00006 class VulkanImageView {
00007 public:
00008     VulkanImageView();
00009
00010     void setImageView(VkImageView imageView);
00011
00012     VkImageView& getImageView() { return imageView; };
00013
00014     void create(VulkanDevice* device, VkImage image, VkFormat format,
00015                 VkImageAspectFlags aspect_flags, uint32_t mip_levels);
00016
00017     void cleanUp();
00018
00019     ~VulkanImageView();
00020
00021 private:
00022     VulkanDevice* device{VK_NULL_HANDLE};
00023
00024     VkImageView imageView;
00025 };
```

## 6.111 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngine ← Vulkan/include/vulkan\_base/VulkanInstance.hpp File Reference

### Data Structures

- class [VulkanInstance](#)

## 6.112 VulkanInstance.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002 #include <vulkan/vulkan.h>
00003 #define GLFW_INCLUDE_NONE
00004 #define GLFW_INCLUDE_VULKAN
00005 #include <GLFW/glfw3.h>
00006
00007 #include <vector>
00008
00009 #include "VulkanDebug.hpp"
00010
00011 class VulkanInstance {
00012 public:
00013     VulkanInstance();
00014
00015     VkInstance& getVulkanInstance() { return instance; };
00016
00017     void cleanUp();
00018
00019     ~VulkanInstance();
00020
00021 private:
00022     VkInstance instance;
00023
00024     // use the standard validation layers from the SDK for error checking
00025     std::vector<const char*> validationLayers = {"VK_LAYER_KHRONOS_validation"};
00026
00027     bool check_validation_layer_support();
00028     bool check_instance_extension_support(
00029         std::vector<const char**> check_extensions);
00030 };
```

## 6.113 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngine Vulkan/include/vulkan\_base/VulkanSwapChain.hpp File Reference

### Data Structures

- class [VulkanSwapChain](#)

## 6.114 VulkanSwapChain.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002
00003 #include "Texture.hpp"
00004 #include "Utilities.hpp"
00005 #include "VulkanDevice.hpp"
00006 #include "Window.hpp"
00007
00008 class VulkanSwapChain {
00009 public:
00010     VulkanSwapChain();
00011
00012     void initVulkanContext(VulkanDevice* device, Window* window,
00013                             const VkSurfaceKHR& surface);
00014
00015     const VkSwapchainKHR& getSwapChain() const { return swapchain; }
00016     uint32_t getNumberOfSwapChainImages() const {
00017         return static_cast<uint32_t>(swap_chain_images.size());
00018     }
00019     const VkExtent2D& getSwapChainExtent() const { return swap_chain_extent; }
00020     const VkFormat& getSwapChainFormat() const {
00021         return swap_chain_image_format;
00022     }
00023     Texture& getSwapChainImage(uint32_t index) {
00024         return swap_chain_images[index];
00025     }
00026
00027     void cleanUp();
00028
00029 ~VulkanSwapChain();
00030
00031 private:
00032     VulkanDevice* device{VK_NULL_HANDLE};
00033     Window* window{VK_NULL_HANDLE};
00034
00035     VkSwapchainKHR swapchain{VK_NULL_HANDLE};
00036
00037     std::vector<Texture> swap_chain_images;
00038     VkFormat swap_chain_image_format{VK_FORMAT_B8G8R8A8_UNORM};
00039     VkExtent2D swap_chain_extent{0, 0};
00040
00041     VkSurfaceFormatKHR chooseBestSurfaceFormat(
00042         const std::vector<VkSurfaceFormatKHR>& formats);
00043     VkPresentModeKHR chooseBestPresentationMode(
00044         const std::vector<VkPresentModeKHR>& presentation_modes);
00045     VkExtent2D chooseSwapExtent(
00046         const VkSurfaceCapabilitiesKHR& surface_capabilities);
00047 };
```

## 6.115 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngine Vulkan/include/window/Window.hpp File Reference

### Data Structures

- class [Window](#)

## 6.116 Window.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <stdio.h>
00003
00004 #define GLFW_INCLUDE_NONE
00005 #define GLFW_INCLUDE_VULKAN
00006
00007 #include <GLFW/glfw3.h>
00008
00009 class Window {
00010 public:
00011     Window();
00012     Window(uint32_t window_width, uint32_t window_height);
00013
00014     // init glfw and its context ...
00015     int initialize();
00016     void cleanUp();
00017
00018     // GETTER functions
00019     bool get_should_close() { return glfwWindowShouldClose(main_window); }
00020     float get_buffer_width() const { return (float)window_buffer_width; }
00021     float get_buffer_height() const { return (float)window_buffer_height; }
00022     float get_x_change();
00023     float get_y_change();
00024     GLFWwindow* get_window() { return main_window; }
00025
00026     float get_height();
00027     float get_width();
00028
00029     bool* get_keys() { return keys; }
00030     bool framebuffer_size_has_changed();
00031     void reset_framebuffer_has_changed();
00032
00033     // SETTER functions
00034     void update_viewport();
00035     void set_buffer_size(float window_buffer_width, float window_buffer_height);
00036
00037     ~Window();
00038
00039 private:
00040     GLFWwindow* main_window;
00041     uint32_t window_width, window_height;
00042     // what key(-s) was/were pressed
00043     bool keys[1024];
00044     float last_x;
00045     float last_y;
00046     float x_change;
00047     float y_change;
00048     bool mouse_first_moved;
00049     bool framebuffer_resized;
00050
00051     // buffers to store our window data to
00052     int window_buffer_width, window_buffer_height;
00053
00054     // we need to start our window callbacks for interaction
00055     void init_callbacks();
00056     static void framebuffer_size_callback(GLFWwindow* window, int width,
00057                                              int height);
00058
00059     // need to be static ...
00060     static void key_callback(GLFWwindow* window, int key, int code, int action,
00061                             int mode);
00062     static void mouse_callback(GLFWwindow* window, double x_pos, double y_pos);
00063     static void mouse_button_callback(GLFWwindow* window, int button, int action,
00064                                     int mods);
00065 };

```

## 6.117 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/brdf/disney.glsl File Reference

## 6.118 disney.glsl

[Go to the documentation of this file.](#)

```

00001 #include "../common/ShadingLibrary.glsl"
00002
00003 // the famous disney principled brdf model
00004 // based on their paper
00005 // (https://blog.selfshadow.com/publications/s2012-shading-course/burley/s2012\_pbs\_disney\_brdf\_notes\_v3.pdf)
00006 // brent burley even released some helpful shader code:
00007 // https://github.com/wdas/brdf/blob/main/src/brdfs/disney.brdf
00008 // (https://schuttejoe.github.io/post/disneybsdf/)
00009 // (https://blog.selfshadow.com/publications/s2015-shading-course/burley/s2015\_pbs\_disney\_bsdf\_notes.pdf)
00010 vec3 mon2lin(vec3 x) {
00011
00012     // https://de.wikipedia.org/wiki/Weber-Fechner-Gesetz
00013     // bring it in linear space when you want to do linear calculations
00014     return vec3(pow(x[0], 2.2), pow(x[1], 2.2), pow(x[2], 2.2));
00015
00016 }
00017
00018 vec3 DisneyDiffuse(vec3 ambient, vec3 L, vec3 V, vec3 N, float roughness, float subsurface) {
00019
00020     vec3 wh = normalize(L) + normalize(V);
00021     wh = normalize(wh);
00022
00023     float cosTheta_d = clamp(dot(normalize(V), wh), -1.0f, 1.0f);
00024     float F_D90 = 0.5f + 2.0f * roughness * cosTheta_d * cosTheta_d;
00025
00026     vec3 lambertDiffuse = LambertDiffuse(ambient);
00027
00028     // even if lit by the backside or viewed from backside
00029     // we want some ambient light!
00030     // catch it here to avoid brutal brightness because of
00031     // following attenuation terms ::)
00032     if (CosTheta(L, N) <= 0 || CosTheta(V, N) <= 0) {
00033         return 0.3f * lambertDiffuse;
00034     }
00035
00036     float F_light = 1.0f + (F_D90 - 1.0f) * pow(1.0f - CosTheta(L, N), 5);
00037     float F_view = 1.0f + (F_D90 - 1.0f) * pow(1.0f - CosTheta(V, N), 5);
00038
00039     // add some "fake subsampling" here
00040     // strictly copied from their shader ...
00041     // Based on Hanrahan-Krueger brdf approximation of isotropic bssrdf
00042     // 1.25 scale is used to (roughly) preserve albedo
00043     // Fss90 used to "flatten" retroreflection based on roughness
00044     float Fss90 = Cos2Theta(L, wh) * roughness;
00045     float Fss = mix(1.0, Fss90, F_light) * mix(1.0, Fss90, F_view);
00046     float ss = 1.25 * (Fss * (1.0 / (CosTheta(L, wh) + CosTheta(V, N)) - .5) + .5);
00047
00048     return lambertDiffuse * mix(F_light * F_view, ss, subsurface);
00049
00050 }
00051
00052 // secondary lobe represents a clearcoat layer over the base: GTR1
00053 // material, and is thus always isotropic and non-metallic.
00054 float D_Disney_Secondary(vec3 wh, vec3 N, float a) {
00055
00056     float alpha2 = a * a;
00057     float denom3 = (1.0f + (alpha2 - 1.0f) * Cos2Theta(wh, N));
00058
00059     return (alpha2 - 1.0f) / (PI * log(alpha2) * denom3);
00060 }
00061
00062 // the D function for their primary layer: GTR2aniso
00063 // represents the base material and may be anisotropic and / or metallic
00064 float D_Disney_Primary(vec3 wh, vec3 N, float roughness, float alphax, float alphay) {
00065
00066     float alphax2 = alphax * alphax;
00067     float alphay2 = alphay * alphay;
00068
00069     float cos2Theta_h = Cos2Theta(wh, N);
00070     float h_dot_y = SinTheta(wh, N) * SinPhi(wh, N);
00071     float h_dot_x = SinTheta(wh, N) * CosPhi(wh, N);
00072     float h_dot_y2 = h_dot_y * h_dot_y;
00073     float h_dot_x2 = h_dot_x * h_dot_x;
00074
00075     float denom3 = (h_dot_x2 / alphax2) + (h_dot_y2 / alphay2) + cos2Theta_h;
00076
00077     return 1.0f / (PI * alphax * alphay * denom3 * denom3);
00078 }
00079
00080 float G_Disney(vec3 wi, vec3 wo, vec3 N, float roughness, float alphax, float alphay)
00081 {
00082     return G2_GGX_SMITH(wi, wo, N, roughness, alphax, alphay);
00083 }
00084
00085 float G_Disney_SmithGGXIso(vec3 wi, vec3 wo, vec3 N, float roughness) {

```

```

00086
00087     return G2_GGX_SMITH_ISOTROPIC(wi, wo, N, roughness);
00088
00089 }
00090
00091 vec3 evaluateDisneyPBR(vec3 ambient, vec3 N, vec3 L, vec3 V, float roughness, vec3 light_color, float
light_intensity) {
00092
00093     // all values in [0,1]
00094
00095     float anisotropic = 0.9;
00096     float metallic = 0.0f;
00097     // subsurface parameter blends between the base diffuse shape and one inspired by the
HanrahanKrueger subsurface BR
00098     // This is useful for giving a subsurface appearance on distant objects
00099     // and on objects where the average scattering path length is small
00100    float subsurface = 0.0f;
00101    float specular = .9f;
00102    // a concession for artistic control that tints incident specular towards the base color.
00103    // Grazing specular is still achromatic.
00104    float specularTint = 0.7;
00105    // an additional grazing(abschrflen) component, primarily intended for cloth; Glanz
00106    float sheen = 0.0;
00107    // amount to tint sheen towards base color.
00108    float sheenTint = 0.4;
00109    // a second, special-purpose specular lobe
00110    float clearcoat = 0.1f;
00111    // clearcoatGloss: controls clearcoat glossiness (0 = a satin appearance, 1 = a gloss appearance)
00112    float clearcoatGloss = 0.5;
00113
00114    // ambient term brought to linear space for calculations
00115    vec3 linAmbient = mon2lin(1.5f * ambient);
00116    // based on number of seecells in the eye; the number of blue cells are the highest amount :))
00117    float luminance = 0.3f*linAmbient[0] + 0.6f*linAmbient[1] + 0.1f*linAmbient[2];
00118    // normalize lum. to isolate hue+sat
00119    vec3 ambientTint = luminance > 0 ? linAmbient/luminance : vec3(1);
00120    // linear map specular coefficient to range [0.0, 0.08]
00121    // this corresponds to IOR values in [1.0, 1.8] -> most common materials
00122    // don't add specular for metals
00123    float remappedSpecular = specular * 0.08f;
00124    vec3 Cspec0 = mix(remappedSpecular * mix(vec3(1), ambientTint, specularTint), linAmbient,
metallic);
00125    vec3 Csheen = mix(vec3(1), ambientTint, sheenTint);
00126
00127
00128    // add diffuse term
00129    // add diffuse term before checking negativ cosinus!
00130    // we want some diffuse light for the sun even if cosinus negative
00131    vec3 diffuse = DisneyDiffuse(linAmbient, L, V, N, roughness, subsurface) * (1.f - metallic) *
CosTheta(L, N);
00132
00133    // 1.) case: get lit by light from the backside
00134    // 2.) case: view it from the back
00135    // you also need to take care of the equal zero case; otherwise divide by zero problems
00136    if (CosTheta(L,N) <= 0 || CosTheta(V,N) <= 0) {
00137        //return diffuse;
00138        return diffuse;
00139    }
00140
00141    //sheen
00142    vec3 Fsheen = pow(1.f - abs(CosTheta(L, N)), 5) * sheen * Csheen;
00143
00144    vec3 SheenColor = Fsheen * (1.f - metallic);
00145
00146    vec3 wh = normalize(L+V);
00147
00148    //clearcoat (ior = 1.5 -> F0 = 0.04)
00149    float Dr = D_Disney_Secondary(wh, N, mix(0.1,0.001,clearcoatGloss));
00150    float Fr = mix(.04, 1.0, pow(1.f - abs(CosTheta(L, N)), 5));
00151    // 0.25: fixed IOR of 1.5; polyurethane
00152    //
00153    float Gr = G_Disney_SmithGGXIso(normalize(V), normalize(L), N, 0.25);
00154
00155    vec3 clearCoatColor = vec3(0.25f * clearcoat * Gr * Fr * Dr);
00156
00157    // mapping to anisotropic alpha's; see paper
00158    float aspect = sqrt(1.f - 0.9f * anisotropic);
00159    float alphax = (roughness * roughness) / aspect;
00160    float alphay = (roughness * roughness) * aspect;
00161
00162    //GTR2aniso
00163    float D = D_Disney_Primary(wh, N, roughness, alphax, alphay);
00164    //smith ggx aniso
00165    float G = G_Disney(normalize(V), normalize(L), N, roughness, alphax, alphay);
00166    // simple fresnel schlick approx
00167    vec3 F = mix(Cspec0, vec3(1), fresnel_schlick(CosTheta(wh, V), ambient, metallic));
00168

```

```

00169     // add specular term
00170     vec3 specular_color = light_color * light_intensity * evaluateCookTorrenceSpecularBRDF(D, G, F,
00171     CosTheta(L, N), CosTheta(V, N)) * CosTheta(L, N);
00172     return diffuse + specular_color + clearCoatColor + SheenColor;
00173 }
```

## 6.119 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/brdf/frostbite.glsI File Reference

### 6.120 frostbite.glsI

[Go to the documentation of this file.](#)

```

00001 #include "../common/ShadingLibrary.glsI"
00002
00003 // source: https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf
00004
00005 vec3 F_Schlick(in vec3 f0, in float f90, in float u)
00006 {
00007     return f0 + (f90 - f0) * pow(1.f - u, 5.f);
00008 }
00009
00010 float FrostbiteDiffuse(float NdotV, float NdotL, float LdotH, float roughness) {
00011
00012     float energyBias = mix(0, 0.5, roughness);
00013     float energyFactor = mix(1.0, 1.0 / 1.51, roughness);
00014     float fd90 = energyBias + 2.0 * LdotH * LdotH * roughness;
00015     vec3 f0 = vec3(1.0f, 1.0f, 1.0f);
00016     float lightScatter = F_Schlick(f0, fd90, NdotL).r;
00017     float viewScatter = F_Schlick(f0, fd90, NdotL).r;
00018
00019     return lightScatter * viewScatter * energyFactor;
00020
00021 }
00022
00023 float V_SmithGGXCorrelated(float NdotL, float NdotV, float alphaG)
00024 {
00025     // Original formulation of G_SmithGGX Correlated
00026     // lambda_v = ( -1 + sqrt ( alphaG2 * (1 - NdotL2 ) / NdotL2 + 1)) * 0.5 f;
00027     // lambda_l = ( -1 + sqrt ( alphaG2 * (1 - NdotV2 ) / NdotV2 + 1)) * 0.5 f;
00028     // G_SmithGGXCorrelated = 1 / (1 + lambda_v + lambda_l );
00029     // V_SmithGGXCorrelated = G_SmithGGXCorrelated / (4.0 f * NdotL * NdotV );
00030
00031     // This is the optimize version
00032     float alphaG2 = alphaG * alphaG;
00033     // Caution : the " NdotL *" and " NdotV *" are explicitely inversed , this is not a mistake .
00034     float Lambda_GGXV = NdotL * sqrt((-NdotV * alphaG2 + NdotV) * NdotV + alphaG2);
00035     float Lambda_GGXL = NdotV * sqrt((-NdotL * alphaG2 + NdotL) * NdotL + alphaG2);
00036
00037     return 0.5f / (Lambda_GGXV + Lambda_GGXL);
00038 }
00039
00040 float D_GGX(float NdotH, float m)
00041 {
00042     // Divide by PI is apply later
00043     float m2 = m * m;
00044     float f = (NdotH * m2 - NdotH) * NdotH + 1;
00045     return m2 / (f * f);
00046 }
00047
00048 vec3 evaluateFrostbitePBR(vec3 ambient, vec3 N, vec3 L, vec3 V, float roughness, vec3 light_color,
    float light_intensity) {
00049
00050
00051     float NdotV = abs(dot(N, V)) + 1e-5f; // avoid artifact
00052     vec3 H = normalize(V + L);
00053     float LdotH = clamp(dot(L, H), 0.0f, 1.0f);
00054     float NdotH = clamp(dot(N, H), 0.0f, 1.0f);
00055     float NdotL = clamp(dot(N, L), 0.0f, 1.0f);
00056
00057     // add lambertian diffuse term vec3(0.f);//
00058     vec3 color = FrostbiteDiffuse(NdotV, NdotL, LdotH, roughness) * (ambient / PI) * NdotL;
00059     //
00060     if (LdotH > 0 && NdotH > 0 && NdotL > 0) {
00061         // Specular BRDF
00062         // renge [0,1] from non-/to metallic
```

```

00064     float reflectence = 0.0f;
00065     vec3 f0 = vec3(0.16f * reflectence * reflectence);
00066     // lets assume that at 90 degrees everything will be reflected
00067     float f90 = 1.f;
00068     vec3 F = F_Schlick(f0, f90, LdotH);
00069     float Vis = V_SmithGGXCorrelated(NdotV, NdotL, roughness);
00070     float D = D_GGX(NdotH, roughness);
00071     color += light_color * light_intensity * (1.f/PI) * evaluateCookTorrenceSpecularBRDF(D, Vis,
00072     F, NdotL, NdotV) * NdotL;
00073 }
00074 return color;
00075 }
```

## 6.121 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/brdf/pbrBook.glsl File Reference

### 6.122 pbrBook.glsl

[Go to the documentation of this file.](#)

```

00001 #include "../common/ShadingLibrary.glsl"
00002
00003 // (https://pbr-book.org/3ed-2018/Reflection_Models/Microfacet_Models)
00004
00005 vec3 F_PBRT(vec3 wi, vec3 wh) {
00006
00007     //assuming dielectrics
00008     float cosThetaI = clamp(dot(normalize(wi), normalize(wh)), 0.0f, 1.0f);
00009     float etaI = 1.00029; // air at sea level
00010     float etaT = 1.5; // water 20 Degrees
00011
00012     //« Potentially swap indices of refraction »
00013     /*bool entering = cosThetaI > 0.f;
00014     if (!entering) {
00015         float aux = etaI;
00016         etaI = etaT;
00017         etaT = aux;
00018         cosThetaI = abs(cosThetaI);
00019     }*/
00020
00021     //« Compute cosThetaT using Snells law »
00022     float sinThetaI = sqrt(max(0.f, 1.f - cosThetaI * cosThetaI));
00023     float sinThetaT = etaI / etaT * sinThetaI;
00024     //« Handle total internal reflection »
00025     if (sinThetaT >= 1)
00026         return vec3(1);
00027
00028     float cosThetaT = sqrt(max(0.f, 1.f - sinThetaT * sinThetaT));
00029
00030     float Rparl = ((etaT * cosThetaI) - (etaI * cosThetaT)) /
00031         ((etaT * cosThetaI) + (etaI * cosThetaT));
00032     float Rperp = ((etaI * cosThetaI) - (etaT * cosThetaT)) /
00033         ((etaI * cosThetaI) + (etaT * cosThetaT));
00034
00035     return vec3((Rparl * Rparl + Rperp * Rperp) / 2.f);
00036
00037 }
00038
00039 float RoughnessToAlpha_PBRT(float roughness) {
00040
00041     roughness = max(roughness, 1e-3);
00042     float x = log(roughness);
00043     return 1.62142f + 0.819955f * x + 0.1734f * x * x + 0.0171201f * x * x * x +
00044         0.000640711f * x * x * x * x;
00045
00046 }
00047
00048 // Normal Distribution function -----
00049 float D_GGX_PBRT(vec3 wh, vec3 N, float roughness) {
00050
00051     float tan2Theta = Tan2Theta(wh, N);
00052     // catch infinity
00053     float infinity = 1.0 / 0.0;
00054     if (tan2Theta == infinity) return 0.;
00055     float alphax = RoughnessToAlpha_PBRT(roughness);
00056     float alphay = RoughnessToAlpha_PBRT(roughness);
00057     const float cos4Theta = Cos2Theta(wh, N) * Cos2Theta(wh, N);
```

```

00058     float e = (Cos2Phi(w, N) / (alphax * alphax) + Sin2Phi(w, N) / (alphay * alphay)) * tan2Theta;
00059     return 1.f / (PI * alphax * alphay * cos4Theta * (1.f + e) * (1.f + e));
00060
00061 }
00062
00063 float Lambda_PBRT(vec3 w, vec3 N, float roughness) {
00064
00065     float absTanTheta = abs(TanTheta(w, N));
00066     // catch infinity
00067     float infinity = 1.0 / 0.0;
00068     if (absTanTheta == infinity) return 0.;
00069     float alphax = RoughnessToAlpha_PBRT(roughness);
00070     float alphay = RoughnessToAlpha_PBRT(roughness);
00071     float alpha = sqrt(Cos2Phi(w, N) * alphax * alphax +
00072         Sin2Phi(w, N) * alphay * alphay);
00073     float alpha2Tan2Theta = (alpha * absTanTheta) * (alpha * absTanTheta);
00074     return (-1.f + sqrt(1.f + alpha2Tan2Theta)) / 2.f;
00075
00076 }
00077
00078 // Geometric Shadowing function -----
00079 float G_GGX_PBRT(vec3 wi, vec3 wo, vec3 N, float roughness) {
00080
00081     return 1.f / (1.f + Lambda_PBRT(wo, N, roughness) + Lambda_PBRT(wi, N, roughness));
00082
00083 }
00084
00085 vec3 evaluatePBRBooksPBR(vec3 ambient, vec3 N, vec3 L, vec3 V, float roughness, vec3 light_color,
00086     float light_intensity) {
00087
00088     // add lambertian diffuse term
00089     vec3 color = LambertDiffuse(ambient) * CosTheta(L, N);
00090
00091     vec3 wo = normalize(L);
00092     vec3 wi = normalize(V);
00093     vec3 wh = normalize(wi + wo);
00094
00095     float cosTheta0 = CosTheta(wo, N);
00096     float cosThetaI = CosTheta(wi, N);
00097     if (cosThetaI <= 0 || cosTheta0 <= 0) return color;
00098     if (wh.x == 0 && wh.y == 0 && wh.z == 0) return color;
00099
00100     if (CosTheta(L, N) > 0 && CosTheta(V, N) > 0) {
00101
00102         // add specular term
00103         float D = D_GGX_PBRT(wh, N, roughness);
00104         float G = G_GGX_PBRT(wi, wo, N, roughness);
00105         vec3 F = F_PBRT(wi, wh);
00106
00107         color += light_color * light_intensity * evaluateCookTorrenceSpecularBRDF(D, G, F, CosTheta(L,
00108             N), CosTheta(V, N)) * CosTheta(L, N);
00109     }
00110 }
```

## **6.123 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/brdf/phong.glsl File Reference**

### **6.124 phong.glsl**

[Go to the documentation of this file.](#)

```

00001 #include "../common/ShadingLibrary.glsl"
00002
00003 //source: Models of Light Reflection for Computer Synthesized Pictures
00004 // https://dl.acm.org/doi/pdf/10.1145/563858.563893
00005 // take normalization term from Akine Mller RealTimeRenderingThirdEdition page 111
00006
00007 vec3 evaluatePhong(vec3 ambient, vec3 N, vec3 L, vec3 V, vec3 light_color, float light_intensity) {
00008
00009     // ignore ambient term for now :
0010     float ka = 0.f;
0011     float ks = 0.3f;
0012
0013     // exponent controls roughness; is in [0,1] because of remapping
0014     float n = 0.5;
0015     // use remapping for better user controllability
```

```

00016 // use the same as in Call of Duty: Black Ops (source RealTimeRenderingThirdEdition page 340)
00017 float maximumSpecular = 8192;
00018 float remappedN = pow(maximumSpecular, ks);
00019
00020 // add lambertian diffuse term
00021 vec3 color = LambertDiffuse(ambient) * light_intensity;
00022 vec3 wh = normalize(L + V);
00023
00024 // add specular term
00025 if (CosTheta(L, N) > 0 && CosTheta(V, N) > 0) {
00026     color += light_color * light_intensity * ks * ((remappedN + 8.f)/(8.f*PI) *
00027         pow(CosTheta(wh,N), remappedN));
00028 }
00029 return color;
00030
00031 }
```

## 6.125 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/brdf/unreal4.glsl File Reference

### 6.126 unreal4.glsl

[Go to the documentation of this file.](#)

```

00001 #include "../common/ShadingLibrary.glsl"
00002
00003 // (https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf)
00004 // https://github.com/SaschaWillems/Vulkan/blob/master/data/shaders/glsl/pbrbasic/pbr.frag
00005 float D_GGX_EPIC_GAMES(vec3 wh, vec3 N, float roughness)
00006 {
00007     float dotNH = clamp(dot(normalize(wh), normalize(N)), 0.0f, 1.0);
00008     float alpha = roughness * roughness;
00009     float alpha2 = alpha * alpha;
00010     float denom = dotNH * dotNH * (alpha2 - 1.0) + 1.0;
00011     return (alpha2) / (PI * denom * denom);
00012 }
00013
00014 float G_GGX_EPIC_GAMES(vec3 wi, vec3 wo, vec3 N, float roughness)
00015 {
00016     float dotNL = clamp(dot(normalize(wo), normalize(N)), 0.0, 1.0);
00017     float dotNV = clamp(dot(normalize(wi), normalize(N)), 0.0, 1.0);
00018     float r = (roughness + 1.0);
00019     float k = (r * r) / 8.0;
00020     float GL = dotNL / (dotNL * (1.0 - k) + k);
00021     float GV = dotNV / (dotNV * (1.0 - k) + k);
00022     return GL * GV;
00023 }
00024
00025 vec3 F_EPIC_GAMES(vec3 wi, vec3 wh, vec3 ambient_color, float metallic) {
00026
00027     float cosTheta = clamp(dot(wi, wh), 0.0f, 1.0f);
00028     vec3 F0 = mix(vec3(0.04), ambient_color, metallic);
00029     vec3 F = F0 + (1.0 - F0) * pow(2, (-5.55473 * cosTheta - 6.98316) * cosTheta);
00030     return F;
00031
00032 }
00033
00034 vec3 evaluateUnreal4PBR(vec3 ambient, vec3 N, vec3 L, vec3 V, float roughness, vec3 light_color, float
light_intensity) {
00035
00036     float cosTheta_l = CosTheta(L, N);
00037     float cosTheta_v = CosTheta(V, N);
00038
00039     // add lambertian diffuse term
00040     vec3 color = LambertDiffuse(ambient) * cosTheta_l;
00041
00042     vec3 wo = normalize(L);
00043     vec3 wi = normalize(V);
00044
00045     float cosThetaO = AbsCosTheta(wo, N);
00046     float cosThetaI = AbsCosTheta(wi, N);
00047     vec3 wh = wi + wo;
00048     if (cosThetaI == 0 || cosThetaO == 0) return vec3(0);
00049     if (wh.x == 0 && wh.y == 0 && wh.z == 0) return vec3(0);
00050     wh = normalize(wh);
00051
00052     float D = D_GGX_EPIC_GAMES(wh, N, roughness);
```

```
00053     float G = G_GGX_EPIC_GAMES(wi, wo, N, roughness);
00054     vec3 F = F_EPIC_GAMES(wi, wh, ambient, .0f);
00055
00056     // add specular term
00057     if (cosTheta_l > 0 && cosTheta_v > 0) {
00058         color += light_color * light_intensity * evaluateCookTorrenceSpecularBRDF(D, G, F, cosTheta_l,
00059         cosTheta_v) * cosTheta_l;
00060     }
00061     return color;
00062
00063 }
```

## 6.127 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/common/Matlib.glsL File Reference

### 6.128 Matlib.glsL

[Go to the documentation of this file.](#)

```
00001 // some common math functions :
00002 #ifndef MATH_LIB
00003 #define MATH_LIB
00004 const float PI = 3.14159265358979323846;
00005
00006 float CosTheta(const vec3 w, vec3 N) {
00007     return dot(w,N);
00008 }
00009
00010 float Cos2Theta(const vec3 w, vec3 N) {
00011     float cosTheta = CosTheta(w,N);
00012     return cosTheta * cosTheta;
00013 }
00014
00015 float AbsCosTheta(const vec3 w, vec3 N) {
00016     return abs(dot(w,N));
00017 }
00018
00019 float Sin2Theta(const vec3 w, vec3 N) {
00020     return max(0.f, 1.f - Cos2Theta(w,N));
00021 }
00022
00023 float SinTheta(const vec3 w, vec3 N) {
00024     return sqrt(Sin2Theta(w,N));
00025 }
00026
00027 float TanTheta(const vec3 w, vec3 N) {
00028     return SinTheta(w,N) / CosTheta(w,N);
00029 }
00030
00031 float Tan2Theta(const vec3 w, vec3 N) {
00032     return Sin2Theta(w,N) / Cos2Theta(w,N);
00033 }
00034
00035 float CosPhi(const vec3 w, vec3 N) {
00036     float sinTheta = SinTheta(w, N);
00037     return (sinTheta == 0) ? 1 : clamp(w.x / sinTheta, -1.f, 1.f);
00038 }
00039 float SinPhi(const vec3 w, vec3 N) {
00040     float sinTheta = SinTheta(w, N);
00041     return (sinTheta == 0) ? 0 : clamp(w.y / sinTheta, -1.f, 1.f);
00042 }
00043
00044 float Cos2Phi(const vec3 w, vec3 N) {
00045     return CosPhi(w,N) * CosPhi(w,N);
00046 }
00047 float Sin2Phi(const vec3 w, vec3 N) {
00048     return SinPhi(w, N) * SinPhi(w,N);
00049 }
00050
00051 #endif
```

## 6.129 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/common/microfacet.glsl File Reference

### 6.130 microfacet.glsl

[Go to the documentation of this file.](#)

```
00001 // microfacet model developed by Torrance-Sparrow
00002 #ifndef MICROFACET
00003 #define MICROFACET
00004 vec3 evaluateCookTorrenceSpecularBRDF(float D, float G, vec3 F, float cosThetaI, float cosThetaO) {
00005     return vec3((D * G * F) / (4.f * cosThetaI * cosThetaO));
00006 }
00007
00008 }
00009 #endif
```

## 6.131 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/common/raycommon.glsl File Reference

### 6.132 raycommon.glsl

[Go to the documentation of this file.](#)

```
00001 struct HitPayload {
00002
00003     vec3 hit_value;
00004
00005 };
```

## 6.133 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/common/ShadingLibrary.glsl File Reference

### 6.134 ShadingLibrary.glsl

[Go to the documentation of this file.](#)

```
00001 #include "Matlib.glsl"
00002 #include "microfacet.glsl"
00003
00004 #ifndef SHADING_LIB
00005 #define SHADING_LIB
00006 vec3 LambertDiffuse(vec3 ambient) {
00007     return ambient / PI;
00008 }
00009
00010 }
00011
00012 // ref: https://hal.inria.fr/hal-01024289/
00013 // Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs
00014 // by Eric Heitz
00015 // equation (86)
00016 float GGX_SMITH_LAMBDA(vec3 v, vec3 N, float roughness, float alphax, float alphay) {
00017
00018     float alphax2 = alphax * alphax;
00019     float alphay2 = alphay * alphay;
00020     //equation (80)
00021     float alpha_0 = sqrt(Cos2Phi(v,N) * alphax2 + Sin2Phi(v, N) * alphay2);
00022     float a = 1.f / (alpha_0 * TanTheta(v,N));
00023
00024     return (-1.f + sqrt(1.f + (1.f / pow(a,2)))) / 2.f;
00025 }
00026 }
```

```
00027
00028 // ref: https://hal.inria.fr/hal-01024289/
00029 // Understanding the Masking-Shading Function in Microfacet-Based BRDFs
00030 // by Eric Heitz
00031 // equation (43)
00032 float G1_GGX_SMITH(vec3 v, vec3 N, float roughness, float alphax, float alphay) {
00033     return 1.f / (1.f + GGX_SMITH_LAMBDA(v,N, roughness, alphax, alphay));
00035
00036 }
00037
00038 // ref: https://hal.inria.fr/hal-01024289/
00039 // Understanding the Masking-Shading Function in Microfacet-Based BRDFs
00040 // by Eric Heitz
00041 // equation (55)
00042 float G2_GGX_SMITH(vec3 wi, vec3 wo, vec3 N, float roughness, float alphax, float alphay) {
00043     if (dot(wi, N) <= 0.f || dot(wo, N) <= 0.f) return 0.f;
00045     return G1_GGX_SMITH(wi, N, roughness, alphax, alphay) * G1_GGX_SMITH(wo, N, roughness, alphax,
00046         alphay);
00046 }
00047 }
00048
00049 // ref: https://hal.inria.fr/hal-01024289/
00050 // Understanding the Masking-Shading Function in Microfacet-Based BRDFs
00051 // by Eric Heitz
00052 // equation (72)
00053 float GGX_SMITH_LAMBDA_ISOTROPIC(vec3 v, vec3 N, float roughness) {
00054     float a = 1.f / (roughness * TanTheta(v, N));
00055
00056     return (-1.f + sqrt(1.f + (1.f / pow(a, 2)))) / 2.f;
00058
00059 }
00060
00061 // ref: https://hal.inria.fr/hal-01024289/
00062 // Understanding the Masking-Shading Function in Microfacet-Based BRDFs
00063 // by Eric Heitz
00064 // equation (43)
00065 float G1_GGX_SMITH_ISOTROPIC(vec3 wi, vec3 N, float roughness) {
00066     return 1.f / (1.f + GGX_SMITH_LAMBDA_ISOTROPIC(wi, N, roughness));
00068
00069 }
00070
00071 // ref: https://hal.inria.fr/hal-01024289/
00072 // Understanding the Masking-Shading Function in Microfacet-Based BRDFs
00073 // by Eric Heitz
00074 // equation (55)
00075 float G2_GGX_SMITH_ISOTROPIC(vec3 wi, vec3 wo, vec3 N, float roughness) {
00076     if (dot(wi, N) <= 0.f || dot(wo, N) <= 0.f) return 0.f;
00078     return G1_GGX_SMITH_ISOTROPIC(wi, N, roughness) * G1_GGX_SMITH_ISOTROPIC(wo, N, roughness);
00079
00080 }
00081
00082
00083 vec3 fresnel_schlick(float cosTheta, vec3 ambient_color, float metallic) {
00084
00085     vec3 F0 = mix(vec3(0.04), ambient_color, metallic);
00086     vec3 F = F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
00087     return F;
00088
00089 }
00090 #endif
```

## 6.135 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/hostDevice/host\_device\_shared\_vars.hpp File Reference

### Variables

- const int **MAX\_TEXTURE\_COUNT** = 24

#### 6.135.1 Variable Documentation

### 6.135.1.1 MAX\_TEXTURE\_COUNT

```
const int MAX_TEXTURE_COUNT = 24
```

Definition at line 5 of file [host\\_device\\_shared\\_vars.hpp](#).

Referenced by [VulkanRenderer::createDescriptorPoolSharedRenderStages\(\)](#), and [VulkanRenderer::createSharedRenderDescriptorSets\(\)](#).

## 6.136 host\_device\_shared\_vars.hpp

[Go to the documentation of this file.](#)

```
00001 #ifndef HOST_DEVICE_SHARED_VARS
00002 #define HOST_DEVICE_SHARED_VARS
00003
00004 #if NDEBUG
00005 const int MAX_TEXTURE_COUNT = 24;
00006 #else
00007 const int MAX_TEXTURE_COUNT = 1;
00008 #endif
00009
00010 // ----- MAIN RENDER DESCRIPTOR SET ----- START (shared between rasterizer and
00011 // raytracer)
00012 #define globalUBO_BINDING 0
00013 #define sceneUBO_BINDING 1
00014 #define OBJECT_DESCRIPTION_BINDING 2
00015 #define TEXTURES_BINDING 3
00016 #define SAMPLER_BINDING 4
00017 // ----- MAIN RENDER DESCRIPTOR SET ----- END
00018
00019 // ----- RAYTRACING BINDING ----- START
00020 #define TLAS_BINDING 0
00021 #define OUT_IMAGE_BINDING 1
00022 // ----- RAYTRACING BINDING ----- END
00023
00024 #endiff
```

## 6.137 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/path\_tracing/path\_tracing.comp File Reference

## 6.138 path\_tracing.comp

[Go to the documentation of this file.](#)

```
00001 #version 460
00002
00003 #extension GL_EXT_ray_query : require
00004 #extension GL_EXT_nonuniform_qualifier : enable
00005 #extension GL_GOOGLE_include_directive : enable
00006 #extension GL_EXT_scalar_block_layout : enable
00007
00008 #extension GL_EXT_shader_explicit_arithmetic_types_int64 : require
00009 #extension GL_EXT_buffer_reference2 : require
00010
00011 #include "../common/raycommon.glsl"
00012
00013 #include "../hostDevice/host_device_shared_vars.hpp"
00014
00015 #include "../brdf/unreal4.glsl"
00016 #include "../brdf/disney.glsl"
00017 #include "../brdf/pbrBook.glsl"
00018 #include "../brdf/phong.glsl"
00019 #include "../brdf/frostbite.glsl"
00020
00021 #include "../../include/renderer/SceneUBO.hpp"
00022 #include "../../include/renderer/GlobalUBO.hpp"
00023 #include "../../include/renderer/pushConstants/PushConstantPathTracing.hpp"
```

```
00024 #include "../../include/scene/ObjMaterial.hpp"
00025 #include "../../include/scene/Vertex.hpp"
00026 #include "../../include/scene/ObjectDescription.hpp"
00027
00028 layout(local_size_x_id = 0, local_size_y_id = 1) in;
00029
00030 layout (set = 0, binding = globalUBO_BINDING) uniform _GlobalUBO {
00031     GlobalUBO globalUBO;
00032 };
00033
00034 layout (set = 0, binding = sceneUBO_BINDING) uniform _SceneUBO {
00035     SceneUBO sceneUBO;
00036 };
00037 layout(set = 0, binding = OBJECT_DESCRIPTION_BINDING, scalar) buffer ObjectDescription_ {
00038     ObjectDescription i[];
00039 } object_description;
00040
00041 layout(set = 0, binding = SAMPLER_BINDING) uniform sampler texture_sampler[MAX_TEXTURE_COUNT];
00042 layout(set = 0, binding = TEXTURES_BINDING) uniform texture2D tex[MAX_TEXTURE_COUNT];
00043
00044 layout(set = 1, binding = TLAS_BINDING) uniform accelerationStructureEXT TLAS;
00045 layout(set = 1, binding = OUT_IMAGE_BINDING, rgba8) uniform image2D image;
00046
00047 layout(buffer_reference, scalar) buffer Vertices {
00048     Vertex v[];
00049 }; // Positions of an object
00050
00051 layout(buffer_reference, scalar) buffer Indices {
00052     ivec3 i[];
00053 }; // Triangle indices
00054
00055 layout(buffer_reference, scalar) buffer MaterialIDs {
00056     int i[];
00057 }; // per triangle material id
00058
00059 layout(buffer_reference, scalar) buffer Materials {
00060     ObjMaterial m[];
00061 }; // all materials of .obj
00062
00063 layout(push_constant) uniform _PushConstantPathTracing {
00064     PushConstantPathTracing pc_ray;
00065 };
00066
00067 // Steps the RNG and returns a floating-point value between 0 and 1 inclusive.
00068 float stepAndOutputRNGFloat(inout uint rngState)
00069 {
00070     // Condensed version of pcg_output_rxs_m_xs_32_32, with simple conversion to floating-point [0,1].
00071     rngState = rngState * 747796405 + 1;
00072     uint word = ((rngState >> (rngState >> 28) + 4)) ^ rngState) * 277803737;
00073     word = (word >> 22) ^ word;
00074     return float(word) / 4294967295.0f;
00075 }
00076
00077 struct HitInfo
00078 {
00079     vec3 color;
00080     vec3 worldPosition;
00081     vec3 worldNormal;
00082 };
00083
00084 HitInfo getObjectHitInfo(rayQueryEXT rayQuery)
00085 {
00086     const int instanceCustomIndex = rayQueryGetIntersectionInstanceCustomIndexEXT(rayQuery, true);
00087     const mat4x3 objToWorld = rayQueryGetIntersectionObjectToWorldEXT(rayQuery, true);
00088
00089     ObjectDescription obj_res = object_description.i[instanceCustomIndex];           // array of all
00090     object descriptions
00091     Indices indices = Indices(obj_res.index_address);                                // array of all
00092     indices
00093     Vertices vertices = Vertices(obj_res.vertex_address);                         // array of all
00094     vertices
00095     MaterialIDs materialIDs = MaterialIDs(obj_res.material_index_address);        // array of per
00096     face material indices
00097     Materials materials = Materials(obj_res.material_address);                   // array of all
00098     materials
00099     array of all materials
00100
00101     HitInfo result;
00102     // Get the ID of the triangle
00103     const int primitiveID = rayQueryGetIntersectionPrimitiveIndexEXT(rayQuery, true);
00104
00105     // Get the indices of the vertices of the triangle
00106     const ivec3 i = indices.i[primitiveID];
00107
00108     // Get the vertices of the triangle
00109     const Vertex v0 = vertices.v[i.x];
00110     const Vertex v1 = vertices.v[i.y];
00111     const Vertex v2 = vertices.v[i.z];
```

```

00106
00107 // Get the barycentric coordinates of the intersection
00108 vec3 barycentrics = vec3(0.0, rayQueryGetIntersectionBarycentricsEXT(rayQuery, true));
00109 barycentrics.x = 1.0 - barycentrics.y - barycentrics.z;
00110
00111 // Compute the coordinates of the intersection
00112 const vec3 objectPos = v0.pos * barycentrics.x + v1.pos * barycentrics.y + v2.pos *
00113 barycentrics.z;
00114 result.worldPosition = vec3(objToWorld * vec4(objectPos, 1.0f));
00115
00116 //compute normal at hit position
00117 const vec3 normal_hit = v0.normal * barycentrics.x + v1.normal * barycentrics.y + v2.normal *
00118 barycentrics.z;
00119 const vec3 world_normal_hit = normalize(vec3(objToWorld * vec4(normal_hit, 1.0f)));
00120 // For the main tutorial, object space is the same as world space:
00121 result.worldNormal = world_normal_hit;
00122
00123 vec2 texture_coordinates = v0.texture_coords * barycentrics.x +
00124 v1.texture_coords * barycentrics.y +
00125 v2.texture_coords * barycentrics.z;
00126
00127 // material id is stored per primitive
00128 vec3 ambient = vec3(0.0f);
00129 int texture_id = materials.m[materialIDs.i[primitiveID]].textureID;
00130 ambient += texture(sampler2D(tex[texture_id], texture_sampler[texture_id]),
00131 texture_coordinates).xyz;
00132 //ambient += materials.m[materialIDs.i[primitiveID]].diffuse;
00133
00134 return result;
00135
00136
00137 void main() {
00138
00139 const uvec2 resolution = uvec2(pc_ray.width, pc_ray.height);
00140 const uvec2 pixel = gl_GlobalInvocationID.xy;
00141
00142 // If the pixel is outside of the image, don't do anything:
00143 if((pixel.x >= resolution.x) || (pixel.y >= resolution.y))
00144 {
00145 return;
00146 }
00147
00148 // State of the random number generator.
00149 uint rngState = resolution.x * pixel.y + pixel.x; // Initial seed
00150
00151 // The sum of the colors of all of the samples.
00152 vec3 summedPixelColor = vec3(0.0);
00153
00154 // Limit the kernel to trace at most 64 samples.
00155 const int NUM_SAMPLES = 32;
00156 for(int sampleIdx = 0; sampleIdx < NUM_SAMPLES; sampleIdx++)
00157 {
00158     // vec4(0,0,0,1) in homogenous coordinates hints that it is the position in the origin
00159     // assumption: origin is the standpoint from the viewer
00160     // the inverse gets us the actual world space position
00161     vec4 rayOrigin = inverse(globalUBO.view) * vec4(0, 0, 0, 1);
00162     // do not forget to invert the y-coord since we are in vulkan
00163     const vec2 randomPixelCenter = vec2(pixel) + vec2(stepAndOutputRNGFloat(rngState),
00164 stepAndOutputRNGFloat(rngState));
00165     vec2 randomPixelCenterUV = randomPixelCenter / resolution;
00166     vec2 randomPixelCenterCS = randomPixelCenterUV * 2.0f - 1.0f;
00167
00168     vec4 target = inverse(globalUBO.projection) * vec4(
00169         randomPixelCenterCS.x,
00170         -randomPixelCenterCS.y,
00171         1,
00172         1);
00173
00174     vec4 rayDirection = inverse(globalUBO.view) * vec4(normalize(target.xyz), 0);
00175
00176     vec3 accumulatedRayColor = vec3(1.0); // The amount of light that made it to the end of the
00177     // current ray.
00178
00179     // Limit the kernel to trace at most 32 segments.
00180     for(int tracedSegments = 0; tracedSegments < 2; tracedSegments++)
00181     {
00182         // Trace the ray and see if and where it intersects the scene!
00183         // First, initialize a ray query object:
00184         rayQueryEXT rayQuery;
00185         rayQueryInitializeEXT(rayQuery, // Ray query
00186                               TLAS, // Top-level acceleration structure
00187                               gl_RayFlagsOpaqueEXT, // Ray flags, here saying "treat all geometry
00188                               as opaque"
00189                               0xFF, // 8-bit instance mask, here saying "trace
00190                               against all instances"

```

```

00186             rayOrigin.xyz,           // Ray origin
00187             0.0,                  // Minimum t-value
00188             rayDirection.xyz,      // Ray direction
00189             10000.0);            // Maximum t-value
00190
00191     // Start traversal, and loop over all ray-scene intersections. When this finishes,
00192     // rayQuery stores a "committed" intersection, the closest intersection (if any).
00193     while(rayQueryProceedEXT(rayQuery))
00194     {
00195     }
00196
00197     // Get the type of committed (true) intersection - nothing, a triangle, or
00198     // a generated object
00199     if(rayQueryGetIntersectionTypeEXT(rayQuery, true) ==
00200         gl_RayQueryCommittedIntersectionTriangleEXT) {
00201         // Ray hit a triangle
00202         HitInfo hitInfo = getObjectHitInfo(rayQuery);
00203
00204         // Apply color absorption
00205         accumulatedRayColor *= hitInfo.color;
00206
00207         // Flip the normal so it points against the ray direction:
00208         hitInfo.worldNormal = faceforward(hitInfo.worldNormal, rayDirection.xyz,
00209             hitInfo.worldNormal);
00210
00211         // Start a new ray at the hit position, but offset it slightly along the normal:
00212         rayOrigin = vec4(hitInfo.worldPosition + 0.0001 * hitInfo.worldNormal, 1.0f);
00213
00214         // For a random diffuse bounce direction, we follow the approach of
00215         // Ray Tracing in One Weekend, and generate a random point on a sphere
00216         // of radius 1 centered at the normal. This uses the random_unit_vector
00217         // function from chapter 8.5:
00218         const float theta  = 6.2831853 * stepAndOutputRNGFloat(rngState); // Random in [0,
00219         2pi]
00220         const float u      = 2.0 * stepAndOutputRNGFloat(rngState) - 1.0; // Random in [-1,
00221         1]
00222         const float r      = sqrt(1.0 - u * u);
00223         rayDirection.xyz  = hitInfo.worldNormal + vec3(r * cos(theta), r * sin(theta), u);
00224
00225         } else {
00226
00227             // Ray hit the sky
00228             //accumulatedRayColor *= pc_ray.clearColor.xyz;
00229
00230             // Sum this with the pixel's other samples.
00231             // (Note that we treat a ray that didn't find a light source as if it had
00232             // an accumulated color of (0, 0, 0)).
00233             summedPixelColor += accumulatedRayColor;
00234
00235             break;
00236         }
00237     imageStore(image, ivec2(pixel), vec4(summedPixelColor / float(NUM_SAMPLES), 1.0));
00238 }
00239 }
```

## 6.139 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/post/post.frag File Reference

### 6.140 post.frag

[Go to the documentation of this file.](#)

```

00001 #version 460
00002
00003 #extension GL_GOOGLE_include_directive : enable
00004
00005 #include "../../include/renderer/pushConstants/PushConstantPost.hpp"
00006
00007 layout(location = 0) in vec2 outUV;
00008 layout(location = 0) out vec4 fragColor;
00009
00010 layout(set = 0, binding = 0) uniform sampler2D noisyTxt;
00011
00012 layout(push_constant) uniform _PushConstantPost {
```

```

00013     PushConstantPost pc_post;
00014 };
00015
00016 void main()
00017 {
00018     vec2 uv      = outUV;
00019     float gamma = 1. / 2.2;
00020
00021     vec3 color = texture(noisyTxt, uv).rgb;
00022     //reinhardts tonemapping
00023     vec3 tonemapped_color = color / (color + vec3(1.f));
00024
00025     fragColor    = vec4(pow(tonemapped_color, vec3(gamma)), 1.0f);
00026
00027 }

```

## 6.141 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/post/post.frag.log.txt File Reference

## 6.142 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/post/post.vert File Reference

### 6.143 post.vert

[Go to the documentation of this file.](#)

```

00001 #version 460
00002 layout(location = 0) out vec2 outUV;
00003
00004
00005 out gl_PerVertex
00006 {
00007     vec4 gl_Position;
00008 };
00009
00010
00011 void main()
00012 {
00013     outUV      = vec2((gl_VertexIndex << 1) & 2, gl_VertexIndex & 2);
00014     gl_Position = vec4(outUV * 2.0f - 1.0f, 1.0f, 1.0f);
00015 }

```

## 6.144 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/post/post.vert.log.txt File Reference

## 6.145 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/rasterizer/shader.frag File Reference

### 6.146 shader.frag

[Go to the documentation of this file.](#)

```

00001 #version 460
00002
00003 //#extension GL_ARB_separate_shader_objects : enable
00004 #extension GL_EXT_nonuniform_qualifier : enable
00005 #extension GL_EXT_scalar_block_layout : enable
00006 #extension GL_GOOGLE_include_directive : enable

```

```

00007
00008 #extension GL_EXT_shader_explicit_arithmetic_types_int64 : require
00009 #extension GL_EXT_buffer_reference2 : require
00010
00011 #include "../common/raycommon.glsl"
00012
00013 #include "../hostDevice/host_device_shared_vars.hpp"
00014
00015 #include "../brdf/unreal4.glsl"
00016 #include "../brdf/disney.glsl"
00017 #include "../brdf/pbrBook.glsl"
00018 #include "../brdf/phong.glsl"
00019 #include "../brdf/frostbite.glsl"
00020
00021 #include "../../../../../include/renderer/SceneUBO.hpp"
00022 #include "../../../../../include/scene/ObjMaterial.hpp"
00023 #include "../../../../../include/scene/Vertex.hpp"
00024 #include "../../../../../include/scene/ObjectDescription.hpp"
00025
00026 layout (location = 0) in vec2 texture_coordinates;
00027 layout (location = 1) in vec3 shading_normal;
00028 layout (location = 2) in vec3 fragment_color;
00029 layout (location = 3) in vec3 worldPosition;
00030
00031 layout (set = 0, binding = sceneUBO_BINDING) uniform _SceneUBO {
00032     SceneUBO sceneUBO;
00033 };
00034
00035 layout(set = 0, binding = OBJECT_DESCRIPTION_BINDING, scalar) buffer ObjectDescription_ {
00036     ObjectDescription i[];
00037 } object_description;
00038
00039 layout(buffer_reference, scalar) buffer Vertices {
00040     Vertex v[];
00041 }; // Positions of an object
00042
00043 layout(buffer_reference, scalar) buffer Indices {
00044     ivec3 i[];
00045 }; // Triangle indices
00046
00047 layout(buffer_reference, scalar) buffer MaterialIDs {
00048     int i[];
00049 }; // per triangle material id
00050
00051 layout(buffer_reference, scalar) buffer Materials {
00052     ObjMaterial m[];
00053 }; // all materials of .obj
00054
00055 layout(set = 0, binding = SAMPLER_BINDING) uniform sampler texture_sampler[MAX_TEXTURE_COUNT];
00056 layout(set = 0, binding = TEXTURES_BINDING) uniform texture2D tex[MAX_TEXTURE_COUNT];
00057
00058 layout (location = 0) out vec4 out_color;
00059
00060 void main() {
00061
00062
00063     ObjectDescription obj_res      = object_description.i[0];
00064     // for now only one object allowed :
00065     MaterialIDs materialIDs       = MaterialIDs(obj_res.material_index_address); // material id
00066     per triangle (face)
00067     Materials materials          = Materials(obj_res.material_address);
00068     // array of all materials
00069
00070     vec3 L = normalize(vec3(-sceneUBO.light_dir));
00071     vec3 N = normalize(shading_normal);
00072     vec3 V = normalize(sceneUBO.cam_pos.xyz - worldPosition);
00073
00074     vec3 ambient = vec3(0.f);
00075
00076     int texture_id   = materials.m[materialIDs.i[gl_PrimitiveID]].textureID;
00077     ambient         += texture(sampler2D(tex[texture_id], texture_sampler[texture_id]),
00078                               texture_coordinates).xyz;
00079     //ambient           += materials.m[materialIDs.i[gl_PrimitiveID]].diffuse;
00080
00081     float roughness = 0.9;
00082     vec3 light_color = vec3(1.f);
00083     float light_intensity = 1.0f;
00084
00085     vec3 color = vec3(0);
00086     // mode : switching between PBR models
00087     // [0] --> EPIC GAMES
00088     // [1] --> PBR BOOK
00089     // [2] --> DISNEYS PRINCIPLED
00090     // [3] --> PHONG
00091     // [4] --> FROSTBITE
00092     int mode = 1;
00093     switch (mode) {

```

```

00090     case 0: color += evaluateUnreal4PBR(ambient, N, L, V, roughness, light_color, light_intensity);
00091         break;
00092     case 1: color += evaluatePBRBooksPBR(ambient, N, L, V, roughness, light_color,
00093         light_intensity);
00094         break;
00095     case 2: color += evaluateDisneysPBR(ambient, N, L, V, roughness, light_color,
00096         light_intensity);
00097         break;
00098     case 3: color += evaluatePhong(ambient, N, L, V, light_color, light_intensity);
00099         break;
00100     case 4: color += evaluateFrostbitePBR(ambient, N, L, V, roughness, light_color,
00101         light_intensity);
00102         break;
00103     }
00104 }
```

## 6.147 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/rasterizer/shader.frag.log.txt File Reference

## 6.148 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/rasterizer/shader.vert File Reference

### 6.149 shader.vert

[Go to the documentation of this file.](#)

```

00001 // #extension GL_KHR_vulkan_glsl : enable
00002
00003 #version 460
00004 #extension GL_ARB_separate_shader_objects : enable
00005 #extension GL_EXT_scalar_block_layout : enable
00006 #extension GL_GOOGLE_include_directive : enable
00007
00008 #extension GL_EXT_shader_explicit_arithmetic_types_int64 : require
00009 #extension GL_EXT_buffer_reference2 : require
00010
00011 #include "../common/raycommon.glsl"
00012
00013 #include "../hostDevice/host_device_shared_vars.hpp"
00014
00015 #include "../../../../../include/renderer/GlobalUBO.hpp"
00016 #include "../../../../../include/renderer/SceneUBO.hpp"
00017
00018 #include "../../../../../include/renderer/pushConstants/PushConstantRasterizer.hpp"
00019
00020 layout (location = 0) in vec3 positions;
00021 layout (location = 1) in vec3 normal;
00022 layout (location = 2) in vec3 color;
00023 layout (location = 3) in vec2 tex_coords;
00024
00025 layout (set = 0, binding = globalUBO_BINDING) uniform _GlobalUBO {
00026     GlobalUBO globalUBO;
00027 };
00028
00029 layout (set = 0, binding = sceneUBO_BINDING) uniform _SceneUBO {
00030     SceneUBO sceneUBO;
00031 };
00032
00033 layout (push_constant) uniform _PushConstantRasterizer {
00034     PushConstantRasterizer pc_raster;
00035 };
00036
00037 layout (location = 0) out vec2 texture_coordinates;
00038 layout (location = 1) out vec3 shading_normal;
00039 layout (location = 2) out vec3 fragment_color;
00040 layout (location = 3) out vec3 worldPosition;
00041
00042 out gl_PerVertex
00043 {
00044     vec4 gl_Position;
00045 };
```

## 6.150

C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/rasterizer/shader.vert.log.txt

### File Reference

419

```
00046
00047 void main () {
00048
00049     // -- WE ARE CALCULATION THE MVP WITH THE GLM LIBRARY WHO IS DESIGNED FOR OPENGL
00050     // -- THEREFORE TAKE THE DIFFERENT COORDINATE SYSTEMS INTO ACCOUNT
00051     vec4 opengl_position = globalUBO.projection * globalUBO.view * pc_raster.model *
00052         vec4(positions, 1.0f);
00053     vec4 vulkan_position = vec4(opengl_position.x, -opengl_position.y, opengl_position.z,
00054         opengl_position.w);
00055     worldPosition = vec3(pc_raster.model * vec4(positions, 1.0f));
00056     worldPosition.y *= -1;
00057     shading_normal = vec3(transpose(inverse(pc_raster.model)) * vec4(normal, 0.0f));
00058     texture_coordinates = tex_coords;
00059     fragment_color = color;
00060     gl_Position = vulkan_position;
00061
00062
00063 }
```

## 6.150 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/rasterizer/shader.vert.log.txt File Reference

## 6.151 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/raytracing/raytrace.rchit File Reference

## 6.152 raytrace.rchit

[Go to the documentation of this file.](#)

```
00001 #version 460
00002
00003 #extension GL_EXT_ray_tracing : require
00004 #extension GL_EXT_nonuniform_qualifier : enable
00005 #extension GL_GOOGLE_include_directive : enable
00006 #extension GL_EXT_scalar_block_layout : enable
00007
00008 #extension GL_EXT_shader_explicit_arithmetic_types_int64 : require
00009 #extension GL_EXT_buffer_reference2 : require
00010
00011 #include "../common/raycommon.glsl"
00012
00013 #include "../hostDevice/host_device_shared_vars.hpp"
00014
00015 #include "../brdf/unreal4.glsl"
00016 #include "../brdf/disney.glsl"
00017 #include "../brdf/pbrBook.glsl"
00018 #include "../brdf/phong.glsl"
00019 #include "../brdf/frostbite.glsl"
00020
00021 #include "../../../../../include/renderer/SceneUBO.hpp"
00022 #include "../../../../../include/renderer/pushConstants/PushConstantRayTracing.hpp"
00023 #include "../../../../../include/scene/ObjMaterial.hpp"
00024 #include "../../../../../include/scene/Vertex.hpp"
00025 #include "../../../../../include/scene/ObjectDescription.hpp"
00026
00027 hitAttributeEXT vec2 attribs;
00028
00029 layout(location = 0) rayPayloadInEXT HitPayload payload;
00030 layout(location = 1) rayPayloadEXT bool isShadowed;
00031
00032 layout (set = 0, binding = sceneUBO_BINDING) uniform _SceneUBO {
00033     SceneUBO sceneUBO;
00034 };
00035 layout(set = 0, binding = OBJECT_DESCRIPTION_BINDING, scalar) buffer ObjectDescription_ {
00036     ObjectDescription i[];
00037 } object_description;
00038
00039 layout(set = 0, binding = SAMPLER_BINDING) uniform sampler texture_sampler[MAX_TEXTURE_COUNT];
00040 layout(set = 0, binding = TEXTURES_BINDING) uniform texture2D tex[MAX_TEXTURE_COUNT];
00041
00042 layout(set = 1, binding = TLAS_BINDING) uniform accelerationStructureEXT TLAS;
00043
```

```

00044 layout(buffer_reference, scalar) buffer Vertices {
00045     Vertex v[];
00046 } // Positions of an object
00047
00048 layout(buffer_reference, scalar) buffer Indices {
00049     ivec3 i[];
00050 } // Triangle indices
00051
00052 layout(buffer_reference, scalar) buffer MaterialIDs {
00053     int i[];
00054 } // per triangle material id
00055
00056 layout(buffer_reference, scalar) buffer Materials {
00057     ObjMaterial m[];
00058 } // all materials of .obj
00059
00060 layout(push_constant) uniform _PushConstantRay {
00061     PushConstantRaytracing pc_ray;
00062 };
00063
00064 void main() {
00065
00066     ObjectDescription obj_res = object_description.i[gl_InstanceCustomIndexEXT]; // array of all
00067     object_descriptions
00068     Indices indices = Indices(obj_res.index_address); // array of all
00069     indices
00070     Vertices vertices = Vertices(obj_res.vertex_address); // array of all
00071     vertices
00072     MaterialIDs materialIDs = MaterialIDs(obj_res.material_index_address); // array of per
00073     face material indices
00074     Materials materials = Materials(obj_res.material_address); // array of all
00075     material
00076     array of all materials
00077
00078     // indices of closest-hit triangle
00079     ivec3 ind = indices.i[gl_PrimitiveID];
00080
00081     // vertex of closest-hit triangle
00082     Vertex v0 = vertices.v[ind.x];
00083     Vertex v1 = vertices.v[ind.y];
00084     Vertex v2 = vertices.v[ind.z];
00085
00086     const vec3 barycentrics = vec3(1.0f - attribs.x - attribs.y, attribs.x, attribs.y);
00087
00088     // compute coordinate of hit position
00089     const vec3 hit_pos = v0.pos * barycentrics.x + v1.pos * barycentrics.y + v2.pos * barycentrics.z;
00090     const vec3 world_hit_pos = vec3(gl_ObjectToWorldEXT * vec4(hit_pos, 1.0f));
00091
00092     //compute normal at hit position
00093     const vec3 normal_hit = v0.normal * barycentrics.x + v1.normal * barycentrics.y + v2.normal *
00094     barycentrics.z;
00095     const vec3 world_normal_hit = normalize(vec3(gl_ObjectToWorldEXT * vec4(normal_hit, 1.0f)));
00096
00097     vec2 texture_coordinates = v0.texture_coords * barycentrics.x +
00098         v1.texture_coords * barycentrics.y +
00099         v2.texture_coords * barycentrics.z;
00100
00101     // material id is stored per primitive
00102     vec3 ambient = vec3(0.f);
00103     int texture_id = materials.m[materialIDs.i[gl_PrimitiveID]].textureID;
00104     ambient += texture(sampler2D(tex[texture_id], texture_sampler[texture_id]),
00105     texture_coordinates).xyz;
00106     //ambient += materials.m[materialIDs.i[gl_PrimitiveID]].diffuse;
00107
00108     vec3 L = normalize(vec3(-sceneUBO.light_dir));
00109     // no need to normalize
00110     vec3 N = normalize(normal_hit);
00111     vec3 V = normalize(sceneUBO.cam_pos.xyz - hit_pos);
00112
00113     isShaded = true;
00114     if(dot(world_normal_hit, L) > 0) {
00115
00116         float t_min = 0.001;
00117         float t_max = 10000;
00118         vec3 origin = gl_WorldRayOriginEXT + gl_WorldRayDirectionEXT * gl_HitTEXT;
00119         vec3 ray_dir = L;
00120         uint flags = gl_RayFlagsTerminateOnFirstHitEXT | gl_RayFlagsOpaqueEXT |
00121             gl_RayFlagsSkipClosestHitShaderEXT;
00122         traceRayEXT(TLAS, // acceleration structure
00123                     flags, // rayFlags
00124                     0xFF, // cullMask
00125                     0, // sbtRecordOffset
00126                     0, // sbtRecordStride
00127                     1, // missIndex
00128                     origin, // ray origin
00129                     t_min, // ray min range
00130                     ray_dir, // ray direction
00131                     t_max, // ray max range

```

### 6.153

C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rchit.log.txt

#### File Reference

421

```
00123           1           // payload (location = 1)
00124       );
00125   }
00126 }
00127
00128     float roughness = 0.4;
00129     vec3 light_color = vec3(1.f);
00130     float light_intensity = 1.f;
00131
00132     payload.hit_value = ambient;
00133     // mode : switching between PBR models
00134     // [0] --> EPIC GAMES
00135     // [1] --> PBR BOOK
00136     // [2] --> DISNEYS PRINCIPLED
00137     // [3] --> PHONG
00138     // [4] --> FROSTBITE
00139     if(!isShadowed) {
00140         int mode = 4;
00141         switch (mode) {
00142             case 0: payload.hit_value += evaluteUnreal4PBR(ambient, N, L, V, roughness, light_color,
00143                     light_intensity);
00144                 break;
00145             case 1: payload.hit_value += evaluatePBRBooksPBR(ambient, N, L, V, roughness, light_color,
00146                     light_intensity);
00147                 break;
00148             case 2: payload.hit_value += evaluateDisneysPBR(ambient, N, L, V, roughness, light_color,
00149                     light_intensity);
00150                 break;
00151             case 3: payload.hit_value += evaluatePhong(ambient, N, L, V, light_color, light_intensity);
00152                 break;
00153             case 4: payload.hit_value += evaluateFrostbitePBR(ambient, N, L, V, roughness, light_color,
00154                     light_intensity);
00155                 break;
00156         }
00157     }
```

### 6.153 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/raytracing/raytrace.rchit.log.txt File Reference

### 6.154 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/raytracing/raytrace.rgen File Reference

### 6.155 raytrace.rgen

[Go to the documentation of this file.](#)

```
00001 #version 460
00002
00003 #extension GL_EXT_ray_tracing : require
00004 #extension GL_GOOGLE_include_directive : enable
00005 #extension GL_EXT_shader_explicit_arithmetic_types_int64 : require
00006
00007 #include "../common/raycommon.glsl"
00008
00009 #include "../hostDevice/host_device_shared_vars.hpp"
00010
00011 #include "../../include/renderer/GlobalUBO.hpp"
00012 #include "../../include/renderer/SceneUBO.hpp"
00013 #include "../../include/renderer/pushConstants/PushConstantRayTracing.hpp"
00014
00015 layout(location = 0) rayPayloadEXT HitPayload payload;
00016
00017 layout (set = 0, binding = globalUBO_BINDING) uniform _GlobalUBO {
00018     GlobalUBO globalUBO;
00019 };
00020
00021 layout (set = 0, binding = sceneUBO_BINDING) uniform _SceneUBO {
00022     SceneUBO sceneUBO;
00023 };
00024
00025 layout(set = 1, binding = TLAS_BINDING) uniform accelerationStructureEXT TLAS;
```

```

00026 layout(set = 1, binding = OUT_IMAGE_BINDING, rgba8) uniform image2D image;
00027
00028 layout(push_constant) uniform _PushConstantRay {
00029     PushConstantRaytracing pc_ray;
00030 };
00031
00032 void main() {
00033
00034     const vec2 pixel_center = vec2(gl_LaunchIDEXT.xy) + vec2(0.5);
00035     const vec2 pixel_center_UV = pixel_center / vec2(gl_LaunchSizeEXT.xy);
00036     vec2 pixel_center_cs = pixel_center_UV * 2.0f - 1.0f;
00037
00038     // vec4(0,0,0,1) in homogenous coordinates hints that it is the position in the origin
00039     // assumption: origin is the standpoint from the viewer
00040     // the inverse gets us the actual world space position
00041     vec4 origin = inverse(globalUBO.view) * vec4(0, 0, 0, 1);
00042     // do not forget to invert the y-coord since we are in vulkan
00043     vec4 target = inverse(globalUBO.projection) * vec4(pixel_center_cs.x, -pixel_center_cs.y, 1, 1);
00044     vec4 direction = inverse(globalUBO.view) * vec4(normalize(target.xyz), 0);
00045
00046     uint ray_flags = gl_RayFlagsOpaqueEXT;
00047     float t_min = 0.001;
00048     float t_max = 10000.0;
00049
00050     traceRayEXT(TLAS,
00051         ray_flags,
00052         0xFF, // cull mask
00053         0, // sbt record offset
00054         0, // record stride
00055         0, // miss index
00056         origin.xyz,
00057         t_min,
00058         direction.xyz,
00059         t_max,
00060         0); // payload (location = 0)
00061
00062
00063     imageStore(image, ivec2(gl_LaunchIDEXT.xy), vec4(payload.hit_value, 1.0));
00064     //imageStore(image, ivec2(gl_LaunchIDEXT.xy), vec4(gl_LaunchIDEXT.xy, 0.0, 1.0));
00065
00066 }

```

## 6.156 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/raytracing/raytrace.rgen.log.txt File Reference

## 6.157 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Resources/Shaders/raytracing/raytrace.rmiss File Reference

### 6.158 raytrace.rmiss

[Go to the documentation of this file.](#)

```

00001 #version 460
00002
00003 #extension GL_EXT_ray_tracing : require
00004 #extension GL_GOOGLE_include_directive : enable
00005 #extension GL_EXT_shader_explicit_arithmetic_types_int64 : require
00006
00007 #include "../common/raycommon.glsl"
00008 #include "../../../../../include/renderer/pushConstants/PushConstantRayTracing.hpp"
00009
00010 layout(location = 0) rayPayloadInEXT HitPayload payload;
00011
00012 layout(push_constant) uniform _PushConstantRay{
00013     PushConstantRaytracing pc_ray;
00014 };
00015
00016 void main() {
00017
00018     payload.hit_value = pc_ray.clear_color.xyz * 0.8f;
00019
00020 }

```

6.159  
C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/raytracing/raytrace.rmiss.log.txt  
File Reference 423

**6.159 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/**  
**Resources/Shaders/raytracing/raytrace.rmiss.log.txt File**  
**Reference**

---

**6.160 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/**  
**Resources/Shaders/raytracing/shadow.rmiss File Reference**

## 6.161 shadow.rmiss

[Go to the documentation of this file.](#)

```
00001 #version 460
00002 #extension GL_EXT_ray_tracing : require
00003
00004 layout(location = 1) rayPayloadInEXT bool shadowed;
00005
00006 void main()
00007 {
00008     shadowed = false;
00009 }
```

**6.162 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/**  
**Resources/Shaders/raytracing/shadow.rmiss.log.txt File Reference**

**6.163 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/app/**  
**App.cpp File Reference**

## 6.164 App.cpp

[Go to the documentation of this file.](#)

```
00001 #include "App.hpp"
00002
00003 #include <vulkan/vulkan.h>
00004 #define GLFW_INCLUDE_NONE
00005 #define GLFW_INCLUDE_VULKAN
00006 #include <GLFW/glfw3.h>
00007
00008 #define GLM_FORCE_RADIANS
00009 #define GLM_FORCE_DEPTH_ZERO_TO_ONE
00010
00011 #include <glm/glm.hpp>
00012 #include <glm/mat4x4.hpp>
00013 #include <iostream>
00014 #include <memory>
00015 #include <stdexcept>
00016 #include <vector>
00017
00018 #include "GUI.hpp"
00019 #include "VulkanRenderer.hpp"
00020 #include "Window.hpp"
00021
00022 App::App() {}
00023
00024 int App::run() {
00025     int window_width = 1200;
00026     int window_height = 768;
00027
00028     float delta_time = 0.0f;
00029     float last_time = 0.0f;
00030
00031     std::unique_ptr<Window> window =
```

```

00032     std::make_unique<Window>(window_width, window_height);
00033     std::unique_ptr<Scene> scene = std::make_unique<Scene>();
00034     std::unique_ptr<GUI> gui = std::make_unique<GUI>(window.get());
00035     std::unique_ptr<Camera> camera = std::make_unique<Camera>();
00036
00037     VulkanRenderer vulkan_renderer{window.get(), scene.get(), gui.get(),
00038                                     camera.get()};
00039
00040     while (!window->get_should_close()) {
00041         // poll all events incoming from user
00042         glfwPollEvents();
00043
00044         // handle events for the camera
00045         camera->key_control(window->get_keys(), delta_time);
00046         camera->mouse_control(window->get_x_change(), window->get_y_change());
00047
00048         float now = static_cast<float>(glfwGetTime());
00049         delta_time = now - last_time;
00050         last_time = now;
00051
00052         scene->update_user_input(gui.get());
00053
00054         vulkan_renderer.updateStateDueToUserInput(gui.get());
00055         vulkan_renderer.updateUniforms(scene.get(), camera.get(), window.get());
00056
00057         //// retrieve updates from the UI
00058         gui->render();
00059
00060         vulkan_renderer.drawFrame();
00061     }
00062
00063     vulkan_renderer.finishAllRenderCommands();
00064
00065     scene->cleanUp();
00066     gui->cleanUp();
00067     window->cleanUp();
00068     vulkan_renderer.cleanUp();
00069
00070     return EXIT_SUCCESS;
00071 }
00072
00073 App::~App() {}

```

## 6.165 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/gui/↔ GUI.cpp File Reference

### 6.166 GUI.cpp

[Go to the documentation of this file.](#)

```

00001 #include "GUI.hpp"
00002
00003 #include "QueueFamilyIndices.hpp"
00004 #include "Utilities.hpp"
00005 #include "VulkanDevice.hpp"
00006
00007 #include "VulkanRendererConfig.hpp"
00008
00009 #include <filesystem>
00010
00011 GUI::GUI(Window* window) { this->window = window; }
00012
00013 void GUI::initializeVulkanContext(VulkanDevice* device,
00014                                     const VkInstance& instance,
00015                                     const VkRenderPass& post_render_pass,
00016                                     const VkCommandPool& graphics_command_pool) {
00017     this->device = device;
00018
00019     create_gui_context(window, instance, post_render_pass);
00020     create_fonts_and_upload(graphics_command_pool);
00021 }
00022
00023 void GUI::render() {
00024     // Start the Dear ImGui frame
00025     ImGui_ImplVulkan_NewFrame();
00026     ImGui_ImplGlfw_NewFrame();
00027     ImGui::NewFrame();
00028

```

```
00029 // ImGui::ShowDemoWindow();
00030
00031 // render your GUI
00032 ImGui::Begin("GUI v1.4.4");
00033
00034 if (ImGui::CollapsingHeader("Hot shader reload")) {
00035     if (ImGui::Button("All shader!"))
00036         guiRendererSharedVars.shader_hot_reload_triggered = true;
00037 }
00038 }
00039
00040 ImGui::Separator();
00041
00042 static int e = 0;
00043 ImGui::RadioButton("Rasterizer", &e, 0);
00044 ImGui::SameLine();
00045 ImGui::RadioButton("Raytracing", &e, 1);
00046 ImGui::SameLine();
00047 ImGui::RadioButton("Path tracing", &e, 2);
00048
00049 switch (e) {
00050     case 0:
00051         guiRendererSharedVars.raytracing = false;
00052         guiRendererSharedVars.pathTracing = false;
00053         break;
00054     case 1:
00055         guiRendererSharedVars.raytracing = true;
00056         guiRendererSharedVars.pathTracing = false;
00057         break;
00058     case 2:
00059         guiRendererSharedVars.raytracing = false;
00060         guiRendererSharedVars.pathTracing = true;
00061         break;
00062 }
00063 // ImGui::Checkbox("Ray tracing", &guiRendererSharedVars.raytracing);
00064
00065 ImGui::Separator();
00066
00067 if (ImGui::CollapsingHeader("Graphic Settings")) {
00068     if (ImGui::TreeNode("Directional Light")) {
00069         ImGui::Separator();
00070         ImGui::SliderFloat("Ambient intensity",
00071                            &guiSceneSharedVars.directional_light_radiance, 0.0f,
00072                            50.0f);
00073         ImGui::Separator();
00074         // Edit a color (stored as ~4 floats)
00075         ImGui::ColorEdit3("Directional Light Color",
00076                           guiSceneSharedVars.directional_light_color);
00077         ImGui::Separator();
00078         ImGui::SliderFloat3("Light Direction",
00079                            &guiSceneSharedVars.directional_light_direction, -1.0f,
00080                            1.0f);
00081
00082         ImGui::TreePop();
00083     }
00084 }
00085
00086 ImGui::Separator();
00087
00088 if (ImGui::CollapsingHeader("GUI Settings")) {
00089     ImGuiStyle& style = ImGui::GetStyle();
00090
00091     if (ImGui::SliderFloat("Frame Rounding", &style.FrameRounding, 0.0f, 12.0f,
00092                           "%0.f")) {
00093         style.GrabRounding = style.FrameRounding; // Make GrabRounding always the
00094                                         // same value as FrameRounding
00095     }
00096     {
00097         bool border = (style.FrameBorderSize > 0.0f);
00098         if (ImGui::Checkbox("FrameBorder", &border)) {
00099             style.FrameBorderSize = border ? 1.0f : 0.0f;
00100         }
00101     }
00102     ImGui::SliderFloat("WindowRounding", &style.WindowRounding, 0.0f, 12.0f,
00103                         "%0.f");
00104 }
00105
00106 ImGui::Separator();
00107
00108 if (ImGui::CollapsingHeader("KEY Bindings")) {
00109     ImGui::Text(
00110         "WASD for moving Forward, backward and to the side\nQE for rotating ");
00111 }
00112
00113 ImGui::Separator();
00114 ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
```

```

00116           1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
00117
00118     ImGui::End();
00119 }
00120
00121 void GUI::cleanUp() {
00122   // clean up of GUI stuff
00123   ImGui_ImplVulkan_Shutdown();
00124   ImGui_ImplGlfw_Shutdown();
00125   ImGui::DestroyContext();
00126   vkDestroyDescriptorPool(device->getLogicalDevice(), gui_descriptor_pool,
00127                           nullptr);
00128 }
00129
00130 void GUI::create_gui_context(Window* window, const VkInstance& instance,
00131                             const VkRenderPass& post_render_pass) {
00132   IMGUI_CHECKVERSION();
00133   ImGui::CreateContext();
00134   ImGuiIO& io = ImGui::GetIO();
00135   (void)io;
00136
00137   float size_pixels = 18;
00138
00139   std::stringstream fontDir;
00140   std::filesystem::path cwd = std::filesystem::current_path();
00141   fontDir << cwd.string();
00142   fontDir << RELATIVE_IMGUI_FONTS_PATH;
00143
00144   std::stringstream robo_font;
00145   robo_font << fontDir.str() << "Roboto-Medium.ttf";
00146   std::stringstream Cousine_font;
00147   Cousine_font << fontDir.str() << "Cousine-Regular.ttf";
00148   std::stringstream DroidSans_font;
00149   DroidSans_font << fontDir.str() << "DroidSans.ttf";
00150   std::stringstream Karla_font;
00151   Karla_font << fontDir.str() << "Karla-Regular.ttf";
00152   std::stringstream proggy_clean_font;
00153   proggy_clean_font << fontDir.str() << "ProggyClean.ttf";
00154   std::stringstream proggy_tiny_font;
00155   proggy_tiny_font << fontDir.str() << "ProggyTiny.ttf";
00156
00157   io.Fonts->AddFontFromFileTTF(robo_font.str().c_str(), size_pixels);
00158   io.Fonts->AddFontFromFileTTF(Cousine_font.str().c_str(), size_pixels);
00159   io.Fonts->AddFontFromFileTTF(DroidSans_font.str().c_str(), size_pixels);
00160   io.Fonts->AddFontFromFileTTF(Karla_font.str().c_str(), size_pixels);
00161   io.Fonts->AddFontFromFileTTF(proggy_clean_font.str().c_str(), size_pixels);
00162   io.Fonts->AddFontFromFileTTF(proggy_tiny_font.str().c_str(), size_pixels);
00163
00164   ImGui::PushStyleVar(ImGuiStyleVar_WindowRounding, 10);
00165   ImGui::PushStyleVar(ImGuiStyleVar_FrameRounding, 10);
00166   ImGui::PushStyleVar(ImGuiStyleVar_FrameBorderSize, 1);
00167   io.ConfigFlags |=
00168     ImGuiConfigFlags_NavEnableKeyboard; // Enable Keyboard Controls
00169   io.ConfigFlags |= ImGuiConfigFlags_NavEnableSetMousePos;
00170   io.WantCaptureMouse = true;
00171   // io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad; // Enable Gamepad
00172   // Controls
00173
00174   // Setup Dear ImGui style
00175   ImGui::StyleColorsDark();
00176   // ImGui::StyleColorsClassic();
00177
00178   ImGui_ImplGlfw_InitForVulkan(window->get_window(), false);
00179
00180   // Create Descriptor Pool
00181   VkDescriptorPoolSize gui_pool_sizes[] = {
00182     {VK_DESCRIPTOR_TYPE_SAMPLER, 10},
00183     {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 10},
00184     {VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 10},
00185     {VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 10},
00186     {VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, 10},
00187     {VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, 10},
00188     {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 10},
00189     {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 10},
00190     {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 10},
00191     {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, 10},
00192     {VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, 100}};
00193
00194   VkDescriptorPoolCreateInfo gui_pool_info = {};
00195   gui_pool_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00196   gui_pool_info.flags = VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT;
00197   gui_pool_info.maxSets = 10 * IM_ARRAYSIZE(gui_pool_sizes);
00198   gui_pool_info.poolSizeCount = (uint32_t)IM_ARRAYSIZE(gui_pool_sizes);
00199   gui_pool_info.pPoolSizes = gui_pool_sizes;
00200
00201   VkResult result =
00202     vkCreateDescriptorPool(device->getLogicalDevice(), &gui_pool_info,

```

```

00203             nullptr, &gui_descriptor_pool);
00204     ASSERT_VULKAN(result, "Failed to create a gui descriptor pool!")
00205
00206     QueueFamilyIndices indices = device->getQueueFamilies();
00207
00208     ImGui_ImplVulkan_InitInfo init_info = {};
00209     init_info.Instance = instance;
00210     init_info.PhysicalDevice = device->getPhysicalDevice();
00211     init_info.Device = device->getLogicalDevice();
00212     init_info.QueueFamily = indices.graphics_family;
00213     init_info.Queue = device->getGraphicsQueue();
00214     init_info.DescriptorPool = gui_descriptor_pool;
00215     init_info.PipelineCache = VK_NULL_HANDLE;
00216     init_info.MinImageCount = 2;
00217     init_info.ImageCount = MAX_FRAME_DRAWS;
00218     init_infoAllocator = VK_NULL_HANDLE;
00219     init_info.CheckVkResultFn = VK_NULL_HANDLE;
00220     init_info.Subpass = 0;
00221     init_info.MSAASamples = VK_SAMPLE_COUNT_1_BIT;
00222
00223     ImGui_ImplVulkan_Init(&init_info, post_render_pass);
00224 }
00225
00226 void GUI::create_fonts_and_upload(const VkCommandPool& graphics_command_pool) {
00227     VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
00228         device->getLogicalDevice(), graphics_command_pool);
00229     ImGui_ImplVulkan_CreateFontsTexture(command_buffer);
00230     commandBufferManager.endAndSubmitCommandBuffer(
00231         device->getLogicalDevice(), graphics_command_pool,
00232         device->getGraphicsQueue(), command_buffer);
00233
00234     // wait until no actions being run on device before destroying
00235     vkDeviceWaitIdle(device->getLogicalDevice());
00236     // clear font textures from cpu data
00237     ImGui_ImplVulkan_DestroyFontUploadObjects();
00238 }
00239
00240 GUI::~GUI() {}

```

## 6.167 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/>Main.cpp File Reference

### Functions

- int [main \(\)](#)

#### 6.167.1 Function Documentation

##### 6.167.1.1 [main\(\)](#)

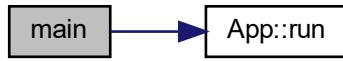
```
int main ( )
```

Definition at line 3 of file [Main.cpp](#).

```
00003     {
00004     App application;
00005     return application.run();
00006 }
```

References [App::run\(\)](#).

Here is the call graph for this function:



## 6.168 Main.cpp

[Go to the documentation of this file.](#)

```

00001 #include "App.hpp"
00002
00003 int main() {
00004     App application;
00005     return application.run();
00006 }
```

## 6.169 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Src/memory/Allocator.cpp File Reference

### 6.170 Allocator.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Allocator.hpp"
00002
00003 #include "Utilities.hpp"
00004
00005 Allocator::Allocator() {}
00006
00007 Allocator::Allocator(const VkDevice& device,
00008                         const VkPhysicalDevice& physicalDevice,
00009                         const VkInstance& instance) {
00010     // see here:
00011     // https://gpuopen-librariesandsdks.github.io/VulkanMemoryAllocator/html/quick_start.html
00012     VmaAllocatorCreateInfo allocatorCreateInfo = {};
00013     allocatorCreateInfo.flags = VMA_ALLOCATOR_CREATE_BUFFER_DEVICE_ADDRESS_BIT;
00014     allocatorCreateInfo.vulkanApiVersion = VK_API_VERSION_1_3;
00015     allocatorCreateInfo.physicalDevice = physicalDevice;
00016     allocatorCreateInfo.device = device;
00017     allocatorCreateInfo.instance = instance;
00018
00019     ASSERT_VULKAN(vmaCreateAllocator(&allocatorCreateInfo, &vmaAllocator),
00020                   "Failed to create vma allocator!")
00021 }
00022
00023 void Allocator::cleanUp() { vmaDestroyAllocator(vmaAllocator); }
00024
00025 Allocator::~Allocator() {}
```

## 6.171 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Src/renderer/accelerationStructures/ASManager.cpp File Reference

### 6.172 ASManager.cpp

[Go to the documentation of this file.](#)

```

00001 #include "ASManager.hpp"
00002
00003 ASManager::ASManager() {}
00004
00005 void ASManager::createASForScene(VulkanDevice* device,
00006                                     VkCommandPool commandPool, Scene* scene) {
00007     this->vulkanDevice = device;
00008     createBLAS(device, commandPool, scene);
00009     createTLAS(device, commandPool, scene);
00010 }
00011
00012 void ASManager::createBLAS(VulkanDevice* device, VkCommandPool commandPool,
00013                             Scene* scene) {
00014     // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
00015     // all functionality from extensions has to be loaded in the beginning
00016     // we need a reference to the device location of our geometry laying on the
00017     // graphics card we already uploaded objects and created vertex and index
00018     // buffers respectively
00019
00020     PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
00021         (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
00022             device->getLogicalDevice(), "vkGetBufferDeviceAddress");
00023
00024     std::vector<BlasInput> blas_input(scene->getModelCount());
00025
00026     for (uint32_t model_index = 0;
00027          model_index < static_cast<uint32_t>(scene->getModelCount());
00028          model_index++) {
00029         std::shared_ptr<Model> mesh_model = scene->get_model_list()[model_index];
00030         // blas_input.emplace_back();
00031         blas_input[model_index].as_geometry.reserve(mesh_model->getMeshCount());
00032         blas_input[model_index].as_build_offset_info.reserve(
00033             mesh_model->getMeshCount());
00034
00035         for (size_t mesh_index = 0; mesh_index < mesh_model->getMeshCount();
00036             mesh_index++) {
00037             VkAccelerationStructureGeometryKHR acceleration_structure_geometry{};
00038             VkAccelerationStructureBuildRangeInfoKHR
00039                 acceleration_structure_build_range_info{};
00040
00041             objectToVkGeometryKHR(device, mesh_model->getMesh(mesh_index),
00042                                   acceleration_structure_geometry,
00043                                   acceleration_structure_build_range_info);
00044             // this only specifies the acceleration structure
00045             // we are building it in the end for the whole model with the build
00046             // command
00047
00048             blas_input[model_index].as_geometry.push_back(
00049                 acceleration_structure_geometry);
00050             blas_input[model_index].as_build_offset_info.push_back(
00051                 acceleration_structure_build_range_info);
00052         }
00053     }
00054
00055     std::vector<BuildAccelerationStructure> build_as_structures;
00056     build_as_structures.resize(scene->getModelCount());
00057
00058     VkDeviceSize max_scratch_size = 0;
00059     VkDeviceSize total_size_all_BLAS = 0;
00060
00061     for (unsigned int i = 0; i < scene->getModelCount(); i++) {
00062         VkDeviceSize current_scratch_size = 0;
00063         VkDeviceSize current_size = 0;
00064
00065         createAccelerationStructureInfosBLAS(device, build_as_structures[i],
00066                                             blas_input[i], current_scratch_size,
00067                                             current_size);
00068
00069         total_size_all_BLAS += current_size;
00070         max_scratch_size = std::max(max_scratch_size, current_scratch_size);
00071     }
00072
00073     VulkanBuffer scratchBuffer;
00074
00075     scratchBuffer.create(device, max_scratch_size,
00076                           VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
00077                           VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00078                           VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00079                           VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
00080                           VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00081
00082     VkBufferDeviceAddressInfo scratch_buffer_device_address_info{};
00083     scratch_buffer_device_address_info.sType =
00084         VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00085     scratch_buffer_device_address_info.buffer = scratchBuffer.getBuffer();
00086
00087     VkDeviceAddress scratch_buffer_address = pvkGetBufferDeviceAddressKHR(

```

```

00088     device->getLogicalDevice(), &scratch_buffer_device_address_info);
00089
00090     VkDeviceOrHostAddressKHR scratch_device_or_host_address{};
00091     scratch_device_or_host_address.deviceAddress = scratch_buffer_address;
00092
00093     VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
00094         device->getLogicalDevice(), commandPool);
00095
00096     for (size_t i = 0; i < scene->getModelCount(); i++) {
00097         createSingleBlas(device, command_buffer, build_as_structures[i],
00098             scratch_buffer_address);
00099
00100     VkMemoryBarrier barrier;
00101     barrier.pNext = nullptr;
00102     barrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
00103     barrier.srcAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
00104     barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR;
00105
00106     vkCmdPipelineBarrier(command_buffer,
00107         VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
00108         VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
00109         0, 1, &barrier, 0, nullptr, 0, nullptr);
00110 }
00111
00112     commandBufferManager.endAndSubmitCommandBuffer(
00113         device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
00114         command_buffer);
00115
00116     for (auto& b : build_as_structures) {
00117         blas.emplace_back(b.single_blas);
00118     }
00119
00120     scratchBuffer.cleanUp();
00121 }
00122
00123 void ASManager::createTLAS(VulkanDevice* device, VkCommandPool commandPool,
00124     Scene* scene) {
00125     // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
00126     // all functionality from extensions has to be loaded in the beginning
00127     // we need a reference to the device location of our geometry laying on the
00128     // graphics card we already uploaded objects and created vertex and index
00129     // buffers respectively
00130     PFN_vkGetAccelerationStructureBuildSizesKHR
00131         pvkGetAccelerationStructureBuildSizesKHR =
00132             (PFN_vkGetAccelerationStructureBuildSizesKHR)vkGetDeviceProcAddr(
00133                 device->getLogicalDevice(),
00134                 "vkGetAccelerationStructureBuildSizesKHR");
00135
00136     PFN_vkCreateAccelerationStructureKHR pvkCreateAccelerationStructureKHR =
00137         (PFN_vkCreateAccelerationStructureKHR)vkGetDeviceProcAddr(
00138             device->getLogicalDevice(), "vkCreateAccelerationStructureKHR");
00139
00140     PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
00141         (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
00142             device->getLogicalDevice(), "vkGetBufferDeviceAddress");
00143
00144     PFN_vkCmdBuildAccelerationStructuresKHR pvkCmdBuildAccelerationStructuresKHR =
00145         (PFN_vkCmdBuildAccelerationStructuresKHR)vkGetDeviceProcAddr(
00146             device->getLogicalDevice(), "vkCmdBuildAccelerationStructuresKHR");
00147
00148     PFN_vkGetAccelerationStructureDeviceAddressKHR
00149         pvkGetAccelerationStructureDeviceAddressKHR =
00150             (PFN_vkGetAccelerationStructureDeviceAddressKHR)vkGetDeviceProcAddr(
00151                 device->getLogicalDevice(),
00152                 "vkGetAccelerationStructureDeviceAddressKHR");
00153
00154     std::vector<VkAccelerationStructureInstanceKHR> tlas_instances;
00155     tlas_instances.reserve(scene->getModelCount());
00156
00157     for (size_t model_index = 0; model_index < scene->getModelCount();
00158         model_index++) {
00159         // glm uses column major matrices so transpose it for Vulkan want row major
00160         // here
00161         glm::mat4 transpose_transform =
00162             glm::transpose(scene->getModelMatrix(static_cast<int>(model_index)));
00163         VkTransformMatrixKHR out_matrix;
00164         memcpy(&out_matrix, &transpose_transform, sizeof(VkTransformMatrixKHR));
00165
00166         VkAccelerationStructureDeviceAddressInfoKHR
00167             acceleration_structure_device_address_info{};
00168         acceleration_structure_device_address_info.sType =
00169             VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_DEVICE_ADDRESS_INFO_KHR;
00170         acceleration_structure_device_address_info.accelerationStructure =
00171             blas[model_index].vulkanAS;
00172
00173         VkDeviceAddress acceleration_structure_device_address =
00174             pvkGetAccelerationStructureDeviceAddressKHR(

```

```
00175         device->getLogicalDevice(),
00176         &acceleration_structure_device_address_info);
00177
00178     VkAccelerationStructureInstanceKHR geometry_instance{};
00179     geometry_instance.transform = out_matrix;
00180     geometry_instance.instanceCustomIndex =
00181         model_index; // gl_InstanceCustomIndexEXT
00182     geometry_instance.mask = 0xFF;
00183     geometry_instance.instanceShaderBindingTableRecordOffset = 0;
00184     geometry_instance.flags =
00185         VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
00186     geometry_instance.accelerationStructureReference =
00187         acceleration_structure_device_address;
00188     geometry_instance.instanceShaderBindingTableRecordOffset =
00189         0; // same hit group for all objects
00190
00191     tlas_instances.emplace_back(geometry_instance);
00192 }
00193
00194 VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
00195     device->getLogicalDevice(), commandPool);
00196
00197 VulkanBuffer geometryInstanceBuffer;
00198
00199 vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00200     device, commandPool, geometryInstanceBuffer,
00201     VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00202         VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00203         VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00204     VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
00205         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
00206     tlas_instances);
00207
00208 VkBufferDeviceAddressInfo geometry_instance_buffer_device_address_info{};
00209 geometry_instance_buffer_device_address_info.sType =
00210     VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00211 geometry_instance_buffer_device_address_info.buffer =
00212     geometryInstanceBuffer.getBuffer();
00213
00214 VkDeviceAddress geometry_instance_buffer_address =
00215     pvkGetBufferDeviceAddressKHR(
00216         device->getLogicalDevice(),
00217         &geometry_instance_buffer_device_address_info);
00218
00219 // Make sure the copy of the instance buffer are copied before triggering the
00220 // acceleration structure build
00221 VkMemoryBarrier barrier;
00222 barrier.pNext = nullptr;
00223 barrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
00224 barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00225 barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
00226 vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00227                         VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
00228                         0, 1, &barrier, 0, nullptr, 0, nullptr);
00229
00230 VkAccelerationStructureGeometryInstancesDataKHR
00231     acceleration_structure_geometry_instances_data{};
00232 acceleration_structure_geometry_instances_data.sType =
00233     VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR;
00234 acceleration_structure_geometry_instances_data.pNext = nullptr;
00235 acceleration_structure_geometry_instances_data.data.deviceAddress =
00236     geometry_instance_buffer_address;
00237
00238 VkAccelerationStructureGeometryKHR topAS_acceleration_structure_geometry{};
00239 topAS_acceleration_structure_geometry.sType =
00240     VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
00241 topAS_acceleration_structure_geometry.pNext = nullptr;
00242 topAS_acceleration_structure_geometry.geometryType =
00243     VK_GEOMETRY_TYPE_INSTANCES_KHR;
00244 topAS_acceleration_structure_geometry.geometry.instances =
00245     acceleration_structure_geometry_instances_data;
00246
00247 // find sizes
00248 VkAccelerationStructureBuildGeometryInfoKHR
00249     acceleration_structure_build_geometry_info{};
00250 acceleration_structure_build_geometry_info.sType =
00251     VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
00252 acceleration_structure_build_geometry_info.pNext = nullptr;
00253 acceleration_structure_build_geometry_info.type =
00254     VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
00255 acceleration_structure_build_geometry_info.flags =
00256     VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
00257 acceleration_structure_build_geometry_info.mode =
00258     VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
00259 acceleration_structure_build_geometry_info.srcAccelerationStructure =
00260     VK_NULL_HANDLE;
00261 acceleration_structure_build_geometry_info.geometryCount = 1;
```

```

00262     acceleration_structure_build_geometry_info.pGeometries =
00263         &topAS_acceleration_structure_geometry;
00264
00265     VkAccelerationStructureBuildSizesInfoKHR
00266         acceleration_structure_build_sizes_info{};
00267     acceleration_structure_build_sizes_info.sType =
00268         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR;
00269     acceleration_structure_build_sizes_info.pNext = nullptr;
00270     acceleration_structure_build_sizes_info.accelerationStructureSize = 0;
00271     acceleration_structure_build_sizes_info.updateScratchSize = 0;
00272     acceleration_structure_build_sizes_info.buildScratchSize = 0;
00273
00274     uint32_t count_instance = static_cast<uint32_t>(tlas_instances.size());
00275     pvkGetAccelerationStructureBuildSizesKHR(
00276         device->getLogicalDevice(), VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_KHR,
00277         &acceleration_structure_build_geometry_info, &count_instance,
00278         &acceleration_structure_build_sizes_info);
00279
00280     // now we got the sizes
00281     VulkanBuffer& tlasVulkanBuffer = tlas.vulkanBuffer;
00282     tlasVulkanBuffer.create(
00283         device, acceleration_structure_build_sizes_info.accelerationStructureSize,
00284         VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR |
00285             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00286             VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00287         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00288             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT);
00289
00290     VkAccelerationStructureCreateInfoKHR acceleration_structure_create_info{};
00291     acceleration_structure_create_info.sType =
00292         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
00293     acceleration_structure_create_info.pNext = nullptr;
00294     acceleration_structure_create_info.createFlags = 0;
00295     acceleration_structure_create_info.buffer = tlasVulkanBuffer.getBuffer();
00296     acceleration_structure_create_info.offset = 0;
00297     acceleration_structure_create_info.size =
00298         acceleration_structure_build_sizes_info.accelerationStructureSize;
00299     acceleration_structure_create_info.type =
00300         VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
00301     acceleration_structure_create_info.deviceAddress = 0;
00302
00303     VkAccelerationStructureKHR& tIAS = tlas.vulkanAS;
00304     pvkCreateAccelerationStructureKHR(device->getLogicalDevice(),
00305                                         &acceleration_structure_create_info,
00306                                         nullptr, &tIAS);
00307
00308     VulkanBuffer scratchBuffer;
00309
00310     scratchBuffer.create(device,
00311         acceleration_structure_build_sizes_info.buildScratchSize,
00312         VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
00313             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00314             VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00315         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00316             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT);
00317
00318     VkBufferDeviceAddressInfo scratch_buffer_device_address_info{};
00319     scratch_buffer_device_address_info.sType =
00320         VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00321     scratch_buffer_device_address_info.buffer = scratchBuffer.getBuffer();
00322
00323     VkDeviceAddress scratch_buffer_address = pvkGetBufferDeviceAddressKHR(
00324         device->getLogicalDevice(), &scratch_buffer_device_address_info);
00325
00326     // update build info
00327     acceleration_structure_build_geometry_info.scratchData.deviceAddress =
00328         scratch_buffer_address;
00329     acceleration_structure_build_geometry_info.srcAccelerationStructure =
00330         VK_NULL_HANDLE;
00331     acceleration_structure_build_geometry_info.dstAccelerationStructure = tIAS;
00332
00333     VkAccelerationStructureBuildRangeInfoKHR
00334         acceleration_structure_build_range_info{};
00335     acceleration_structure_build_range_info.primitiveCount =
00336         scene->getModelCount();
00337     acceleration_structure_build_range_info.primitiveOffset = 0;
00338     acceleration_structure_build_range_info.firstVertex = 0;
00339     acceleration_structure_build_range_info.transformOffset = 0;
00340
00341     VkAccelerationStructureBuildRangeInfoKHR*
00342         acceleration_structure_build_range_infos =
00343             &acceleration_structure_build_range_info;
00344
00345     pvkCmdBuildAccelerationStructuresKHR(
00346         command_buffer, 1, &acceleration_structure_build_geometry_info,
00347         &acceleration_structure_build_range_infos);
00348

```

```

00349     commandBufferManager.endAndSubmitCommandBuffer(
00350         device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
00351         command_buffer);
00352     scratchBuffer.cleanUp();
00353     geometryInstanceBuffer.cleanUp();
00354 }
00355
00356 void ASManager::cleanUp() {
00357     PFN_vkDestroyAccelerationStructureKHR pvkDestroyAccelerationStructureKHR =
00358         (PFN_vkDestroyAccelerationStructureKHR)vkGetDeviceProcAddr(
00359             vulkanDevice->getLogicalDevice(),
00360             "vkDestroyAccelerationStructureKHR");
00361
00362     pvkDestroyAccelerationStructureKHR(vulkanDevice->getLogicalDevice(),
00363                                         tlas.vulkanAS, nullptr);
00364
00365     tlas.vulkanBuffer.cleanUp();
00366
00367     for (size_t index = 0; index < blas.size(); index++) {
00368         pvkDestroyAccelerationStructureKHR(vulkanDevice->getLogicalDevice(),
00369                                         blas[index].vulkanAS, nullptr);
00370
00371         blas[index].vulkanBuffer.cleanUp();
00372     }
00373 }
00374
00375 ASManager::~ASManager() {}
00376
00377 void ASManager::createSingleBlas(
00378     VulkanDevice* device, VkCommandBuffer command_buffer,
00379     BuildAccelerationStructure& build_as_structure,
00380     VkDeviceAddress scratch_device_or_host_address) {
00381     PFN_vkCreateAccelerationStructureKHR pvkCreateAccelerationStructureKHR =
00382         (PFN_vkCreateAccelerationStructureKHR)vkGetDeviceProcAddr(
00383             device->getLogicalDevice(), "vkCreateAccelerationStructureKHR");
00384
00385     PFN_vkCmdBuildAccelerationStructuresKHR pvkCmdBuildAccelerationStructuresKHR =
00386         (PFN_vkCmdBuildAccelerationStructuresKHR)vkGetDeviceProcAddr(
00387             device->getLogicalDevice(), "vkCmdBuildAccelerationStructuresKHR");
00388
00389     VkAccelerationStructureCreateInfoKHR acceleration_structure_create_info{};
00390     acceleration_structure_create_info.sType =
00391         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
00392     acceleration_structure_create_info.type =
00393         VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
00394     acceleration_structure_create_info.size =
00395         build_as_structure.size_info.accelerationStructureSize;
00396     VulkanBuffer& blasVulkanBuffer = build_as_structure.single_blas.vulkanBuffer;
00397     blasVulkanBuffer.create(
00398         device, build_as_structure.size_info.accelerationStructureSize,
00399         VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR |
00400             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00401             VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00402             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
00403             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
00404
00405         acceleration_structure_create_info.buffer = blasVulkanBuffer.getBuffer(),
00406         VkAccelerationStructureKHR& blas_as = build_as_structure.single_blas.vulkanAS,
00407         pvkCreateAccelerationStructureKHR(device->getLogicalDevice(),
00408                                         &acceleration_structure_create_info,
00409                                         nullptr, &blas_as),
00410
00411         build_as_structure.build_info.dstAccelerationStructure = blas_as,
00412         build_as_structure.build_info.scratchData.deviceAddress =
00413             scratch_device_or_host_address,
00414
00415         pvkCmdBuildAccelerationStructuresKHR(command_buffer, 1,
00416                                         &build_as_structure.build_info,
00417                                         &build_as_structure.range_info);
00418 }
00419
00420 void ASManager::createAccelerationStructureInfosBLAS(
00421     VulkanDevice* device, BuildAccelerationStructure& build_as_structure,
00422     BlasInput& blas_input, VkDeviceSize& current_scratch_size,
00423     VkDeviceSize& current_size) {
00424     PFN_vkGetAccelerationStructureBuildSizesKHR
00425         pvkGetAccelerationStructureBuildSizesKHR =
00426             (PFN_vkGetAccelerationStructureBuildSizesKHR)vkGetDeviceProcAddr(
00427                 device->getLogicalDevice(),
00428                 "vkGetAccelerationStructureBuildSizesKHR");
00429
00430     build_as_structure.build_info.sType =
00431         VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
00432     build_as_structure.build_info.type =
00433         VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
00434     build_as_structure.build_info.flags =
00435         VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;

```

```

00436 build_as_structure.build_info.mode =
00437     VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
00438 build_as_structure.build_info.geometryCount =
00439     static_cast<uint32_t>(blas_input.as_geometry.size());
00440 build_as_structure.build_info.pGeometries = blas_input.as_geometry.data();
00441
00442 build_as_structure.range_info = blas_input.as_build_offset_info.data();
00443
00444 build_as_structure.size_info.sType =
00445     VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR;
00446
00447 std::vector<uint32_t> max_primitive_cnt(
00448     blas_input.as_build_offset_info.size());
00449
00450 for (uint32_t temp = 0;
00451     temp < static_cast<uint32_t>(blas_input.as_build_offset_info.size());
00452     temp++)
00453     max_primitive_cnt[temp] =
00454         blas_input.as_build_offset_info[temp].primitiveCount;
00455
00456 pvkGetAccelerationStructureBuildSizesKHR(
00457     device->getLogicalDevice(),
00458     VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR,
00459     &build_as_structure.build_info, max_primitive_cnt.data(),
00460     &build_as_structure.size_info);
00461
00462 current_size = build_as_structure.size_info.accelerationStructureSize;
00463 current_scratch_size = build_as_structure.size_info.buildScratchSize;
00464 }
00465
00466 void ASManager::objectToVkGeometryKHR(
00467     VulkanDevice* device, Mesh* mesh,
00468     VkAccelerationStructureGeometryKHR& acceleration_structure_geometry,
00469     VkAccelerationStructureBuildRangeInfoKHR&
00470         acceleration_structure_build_range_info) {
00471 // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
00472 // all functionality from extensions has to be loaded in the beginning
00473 // we need a reference to the device location of our geometry laying on the
00474 // graphics card we already uploaded objects and created vertex and index
00475 // buffers respectively
00476 PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
00477     (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
00478         device->getLogicalDevice(), "vkGetBufferDeviceAddress");
00479
00480 // all starts with the address of our vertex and index data we already
00481 // uploaded in buffers earlier when loading the meshes/models
00482 VkBufferDeviceAddressInfo vertex_buffer_device_address_info{};
00483 vertex_buffer_device_address_info.sType =
00484     VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00485 vertex_buffer_device_address_info.buffer = mesh->getVertexBuffer();
00486 vertex_buffer_device_address_info.pNext = nullptr;
00487
00488 VkBufferDeviceAddressInfo index_buffer_device_address_info{};
00489 index_buffer_device_address_info.sType =
00490     VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00491 index_buffer_device_address_info.buffer = mesh->getIndexBuffer();
00492 index_buffer_device_address_info.pNext = nullptr;
00493
00494 // receiving address to move on
00495 VkDeviceAddress vertex_buffer_address = pvkGetBufferDeviceAddressKHR(
00496     device->getLogicalDevice(), &vertex_buffer_device_address_info);
00497 VkDeviceAddress index_buffer_address = pvkGetBufferDeviceAddressKHR(
00498     device->getLogicalDevice(), &index_buffer_device_address_info);
00499
00500 // convert to const address for further processing
00501 VkDeviceOrHostAddressConstKHR vertex_device_or_host_address_const{};
00502 vertex_device_or_host_address_const.deviceAddress = vertex_buffer_address;
00503
00504 VkDeviceOrHostAddressConstKHR index_device_or_host_address_const{};
00505 index_device_or_host_address_const.deviceAddress = index_buffer_address;
00506
00507 VkAccelerationStructureGeometryTrianglesDataKHR
00508     acceleration_structure_triangles_data{};
00509 acceleration_structure_triangles_data.sType =
00510     VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR;
00511 acceleration_structure_triangles_data.pNext = nullptr;
00512 acceleration_structure_triangles_data.vertexFormat =
00513     VK_FORMAT_R32G32B32_SFLOAT;
00514 acceleration_structure_triangles_data.vertexData =
00515     vertex_device_or_host_address_const;
00516 acceleration_structure_triangles_data.vertexStride = sizeof(Vertex);
00517 acceleration_structure_triangles_data.maxVertex = mesh->getVertexCount();
00518 acceleration_structure_triangles_data.indexType = VK_INDEX_TYPE_UINT32;
00519 acceleration_structure_triangles_data.indexData =
00520     index_device_or_host_address_const;
00521
00522 // can also be instances or AABBs; not covered here

```

```

00523 // but to identify as triangles put it into these struct
00524 VkAccelerationStructureGeometryDataKHR acceleration_structure_geometry_data{};
00525 acceleration_structure_geometry_data.triangles =
00526     acceleration_structure_triangles_data;
00527
00528 acceleration_structure_geometry.sType =
00529     VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
00530 acceleration_structure_geometry.pNext = nullptr;
00531 acceleration_structure_geometry.geometryType = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
00532 acceleration_structure_geometry.geometry =
00533     acceleration_structure_geometry_data;
00534 acceleration_structure_geometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR;
00535
00536 // we have triangles so divide the number of vertices with 3!!
00537 // for our simple case a no brainer
00538 // take entire data to build BLAS
00539 // number of indices is truly the stick point here
00540 acceleration_structure_build_range_info.primitiveCount =
00541     mesh->getIndexCount() / 3;
00542 acceleration_structure_build_range_info.primitiveOffset = 0;
00543 acceleration_structure_build_range_info.firstVertex = 0;
00544 acceleration_structure_build_range_info.transformOffset = 0;
00545 }

```

## 6.173 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Src/renderer/CommandBufferManager.cpp File Reference

### 6.174 CommandBufferManager.cpp

[Go to the documentation of this file.](#)

```

00001 #include "CommandBufferManager.hpp"
00002
00003 CommandBufferManager::CommandBufferManager() {}
00004
00005 VkCommandBuffer CommandBufferManager::beginCommandBuffer(
00006     VkDevice device, VkCommandPool command_pool) {
00007     // command buffer to hold transfer commands
00008     VkCommandBuffer command_buffer;
00009
00010     // command buffer details
00011     VkCommandBufferAllocateInfo alloc_info{};
00012     alloc_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
00013     alloc_info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
00014     alloc_info.commandPool = command_pool;
00015     alloc_info.commandBufferCount = 1;
00016
00017     // allocate command buffer from pool
00018     vkAllocateCommandBuffers(device, &alloc_info, &command_buffer);
00019
00020     // information to begin the command buffer record
00021     VkCommandBufferBeginInfo begin_info{};
00022     begin_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
00023     // we are only using the command buffer once, so set up for one time submit
00024     begin_info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
00025
00026     // begin recording transfer commands
00027     vkBeginCommandBuffer(command_buffer, &begin_info);
00028
00029     return command_buffer;
00030 }
00031
00032 void CommandBufferManager::endAndSubmitCommandBuffer(
00033     VkDevice device, VkCommandPool command_pool, VkQueue queue,
00034     VkCommandBuffer& command_buffer) {
00035     // end commands
00036     VkResult result = vkEndCommandBuffer(command_buffer);
00037     ASSERT_VULKAN(result, "Failed to end command buffer!")
00038
00039     // queue submission information
00040     VkSubmitInfo submit_info{};
00041     submit_info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
00042     submit_info.commandBufferCount = 1;
00043     submit_info.pCommandBuffers = &command_buffer;
00044
00045     // submit transfer command to transfer queue and wait until it finishes
00046     result = vkQueueSubmit(queue, 1, &submit_info, VK_NULL_HANDLE);
00047     ASSERT_VULKAN(result, "Failed to submit to queue!")

```

```

00048
00049     result = vkQueueWaitIdle(queue);
00050     ASSERT_VULKAN(result, "Failed to wait Idle!")
00051
00052     // free temporary command buffer back to pool
00053     vkFreeCommandBuffers(device, command_pool, 1, &command_buffer);
00054 }
00055
00056 CommandBufferManager::~CommandBufferManager() {}

```

## 6.175 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/← Src/renderer/PathTracing.cpp File Reference

### 6.176 PathTracing.cpp

[Go to the documentation of this file.](#)

```

00001 #include "PathTracing.hpp"
00002
00003 #include <algorithm>
00004 #include <array>
00005 #include <filesystem>
00006
00007 #include "File.hpp"
00008 #include "ShaderHelper.hpp"
00009
00010 #include "VulkanRendererConfig.hpp"
00011
00012 // Good source:
00013 // https://github.com/nvpro-samples/vk_mini_path_tracer/blob/main/vk_mini_path_tracer/main.cpp
00014
00015 PathTracing::PathTracing() {}
00016
00017 void PathTracing::init(
00018     VulkanDevice* device,
00019     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00020     this->device = device;
00021
00022     VkPhysicalDeviceProperties physicalDeviceProps =
00023         device->getPhysicalDeviceProperties();
00024     timeStampPeriod = physicalDeviceProps.limits.timestampPeriod;
00025
00026     // save the limits for handling all special cases later on
00027     computeLimits.maxComputeWorkGroupCount[0] =
00028         physicalDeviceProps.limits.maxComputeWorkGroupCount[0];
00029     computeLimits.maxComputeWorkGroupCount[1] =
00030         physicalDeviceProps.limits.maxComputeWorkGroupCount[1];
00031     computeLimits.maxComputeWorkGroupCount[2] =
00032         physicalDeviceProps.limits.maxComputeWorkGroupCount[2];
00033
00034     computeLimits.maxComputeWorkGroupInvocations =
00035         physicalDeviceProps.limits.maxComputeWorkGroupInvocations;
00036
00037     computeLimits.maxComputeWorkGroupSize[0] =
00038         physicalDeviceProps.limits.maxComputeWorkGroupSize[0];
00039     computeLimits.maxComputeWorkGroupSize[1] =
00040         physicalDeviceProps.limits.maxComputeWorkGroupSize[1];
00041     computeLimits.maxComputeWorkGroupSize[2] =
00042         physicalDeviceProps.limits.maxComputeWorkGroupSize[2];
00043
00044     queryResults.resize(query_count);
00045     createQueryPool();
00046
00047     createPipeline(descriptorSetLayouts);
00048 }
00049
00050 void PathTracing::shaderHotReload(
00051     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
00052     vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
00053     createPipeline(descriptor_set_layouts);
00054 }
00055
00056 void PathTracing::recordCommands(
00057     VkCommandBuffer& commandBuffer, uint32_t image_index,
00058     VulkanImage& vulkanImage, VulkanSwapChain* vulkanSwapChain,
00059     const std::vector<VkDescriptorSet>& descriptorSets) {
00060     // we have reset the pool; hence start by 0
00061     uint32_t query = 0;

```

```

00062
00063     vkCmdResetQueryPool(commandBuffer, queryPool, 0, query_count);
00064
00065     vkCmdWriteTimestamp(
00066         commandBuffer,
00067         VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,
00068         query++);
00069
00070     QueueFamilyIndices indices = device->getQueueFamilies();
00071
00072     VkImageSubresourceRange subresourceRange{};
00073     subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00074     subresourceRange.baseMipLevel = 0;
00075     subresourceRange.baseArrayLayer = 0;
00076     subresourceRange.levelCount = 1;
00077     subresourceRange.layerCount = 1;
00078
00079     VkImageMemoryBarrier presentToPathTracingImageBarrier{};
00080     presentToPathTracingImageBarrier.sType =
00081         VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00082     presentToPathTracingImageBarrier.pNext = nullptr;
00083     presentToPathTracingImageBarrier.srcQueueFamilyIndex =
00084         indices.graphics_family;
00085     presentToPathTracingImageBarrier.dstQueueFamilyIndex = indices.compute_family;
00086     presentToPathTracingImageBarrier.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
00087     presentToPathTracingImageBarrier.dstAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
00088     presentToPathTracingImageBarrier.oldLayout = VK_IMAGE_LAYOUT_GENERAL;
00089     presentToPathTracingImageBarrier.newLayout = VK_IMAGE_LAYOUT_GENERAL;
00090     presentToPathTracingImageBarrier.subresourceRange = subresourceRange;
00091     presentToPathTracingImageBarrier.image = vulkanImage.getImage();
00092
00093     vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,
00094                           VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
00095                           0, 0, nullptr, 0, nullptr, 1,
00096                           &presentToPathTracingImageBarrier);
00097
00098     VkExtent2D imageSize = vulkanSwapChain->getSwapChainExtent();
00099     push_constant.width = imageSize.width;
00100     push_constant.height = imageSize.height;
00101     push_constant.clearColor = {0.2f, 0.65f, 0.4f, 1.0f};
00102
00103     vkCmdPushConstants(commandBuffer, pipeline_layout,
00104                         VK_SHADER_STAGE_COMPUTE_BIT, 0,
00105                         sizeof(PushConstantPathTracing), &push_constant);
00106
00107     vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);
00108
00109     vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE,
00110                             pipeline_layout, 0,
00111                             static_cast<uint32_t>(descriptorSets.size()),
00112                             descriptorSets.data(), 0, 0);
00113
00114
00115     uint32_t workGroupCountX =
00116         std::max((imageSize.width + specializationData.specWorkGroupSizeX - 1) /
00117                  specializationData.specWorkGroupSizeX,
00118                  1U);
00119     uint32_t workGroupCountY =
00120         std::max((imageSize.height + specializationData.specWorkGroupSizeY - 1) /
00121                  specializationData.specWorkGroupSizeY,
00122                  1U);
00123     uint32_t workGroupCountZ = 1;
00124
00125     vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY,
00126                   workGroupCountZ);
00127
00128     VkImageMemoryBarrier pathTracingToPresentImageBarrier{};
00129     pathTracingToPresentImageBarrier.sType =
00130         VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00131     pathTracingToPresentImageBarrier.pNext = nullptr;
00132     pathTracingToPresentImageBarrier.srcQueueFamilyIndex = indices.compute_family;
00133     pathTracingToPresentImageBarrier.dstQueueFamilyIndex =
00134         indices.graphics_family;
00135     pathTracingToPresentImageBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
00136     pathTracingToPresentImageBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
00137     pathTracingToPresentImageBarrier.oldLayout = VK_IMAGE_LAYOUT_GENERAL;
00138     pathTracingToPresentImageBarrier.newLayout = VK_IMAGE_LAYOUT_GENERAL;
00139     pathTracingToPresentImageBarrier.image = vulkanImage.getImage();
00140     pathTracingToPresentImageBarrier.subresourceRange = subresourceRange;
00141
00142     vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
00143                           VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, 0, 0, nullptr, 0,
00144                           nullptr, 1, &pathTracingToPresentImageBarrier);
00145
00146     vkCmdWriteTimestamp(
00147         commandBuffer,
00148         VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,

```

```

00149     query++);
00150     VkResult result = vkGetQueryPoolResults(
00151         device->getLogicalDevice(), queryPool, 0, query_count,
00152         queryResults.size() * sizeof(uint64_t), queryResults.data(),
00153         static_cast<VkDeviceSize>(sizeof(uint64_t)), VK_QUERY_RESULT_64_BIT);
00154
00155     if (result != VK_NOT_READY) {
00156         pathTracingTiming = (static_cast<float>(queryResults[1] - queryResults[0]) *
00157             timeStampPeriod) /
00158             1000000.f;
00159     }
00160 }
00161
00162 void PathTracing::cleanUp() {
00163     vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
00164     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00165
00166     vkDestroyQueryPool(device->getLogicalDevice(), queryPool, nullptr);
00167 }
00168
00169 PathTracing::~PathTracing() {}
00170
00171 void PathTracing::createQueryPool() {
00172     VkQueryPoolCreateInfo queryPoolInfo = {};
00173     queryPoolInfo.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
00174     // This query pool will store pipeline statistics
00175     queryPoolInfo.queryType = VK_QUERY_TYPE_TIMESTAMP;
00176     // Pipeline counters to be returned for this pool
00177     queryPoolInfo.pipelineStatistics =
00178         VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT;
00179     queryPoolInfo.queryCount = query_count;
00180     ASSERT_VULKAN(vkCreateQueryPool(device->getLogicalDevice(), &queryPoolInfo,
00181                                     NULL, &queryPool),
00182                 "Failed to create query pool!");
00183 }
00184
00185 void PathTracing::createPipeline(
00186     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00187     VkPushConstantRange push_constant_range{};
00188     push_constant_range.stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
00189     push_constant_range.offset = 0;
00190     push_constant_range.size = sizeof(PushConstantPathTracing);
00191
00192     VkPipelineLayoutCreateInfo compute_pipeline_layout_create_info{};
00193     compute_pipeline_layout_create_info.sType =
00194         VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00195     compute_pipeline_layout_create_info.setLayoutCount =
00196         static_cast<uint32_t>(descriptorSetLayouts.size());
00197     compute_pipeline_layout_create_info.pushConstantRangeCount = 1;
00198     compute_pipeline_layout_create_info.pPushConstantRanges =
00199         &push_constant_range;
00200     compute_pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00201
00202     ASSERT_VULKAN(vkCreatePipelineLayout(device->getLogicalDevice(),
00203                                         &compute_pipeline_layout_create_info,
00204                                         nullptr, &pipeline_layout),
00205                 "Failed to create compute path tracing pipeline layout!");
00206
00207     // create pipeline
00208     std::stringstream pathTracing_shader_dir;
00209     std::filesystem::path cwd = std::filesystem::current_path();
00210     pathTracing_shader_dir += cwd.string();
00211     pathTracing_shader_dir += RELATIVE_RESOURCE_PATH;
00212     pathTracing_shader_dir += "Shaders/path_tracing/";
00213
00214     std::string pathTracing_shader = "path_tracing.comp";
00215
00216     ShaderHelper shaderHelper;
00217     File pathTracingShaderFile(shaderHelper.getShaderSpvDir(
00218         pathTracing_shader_dir.str(), pathTracing_shader));
00219     std::vector<char> pathTracingShadercode =
00220         pathTracingShaderFile.readCharSequence();
00221
00222     shaderHelper.compileShader(pathTracing_shader_dir.str(), pathTracing_shader);
00223
00224     // build shader modules to link to graphics pipeline
00225     VkShaderModule pathTracingModule =
00226         shaderHelper.createShaderModule(device, pathTracingShadercode);
00227
00228     // Specialization constant for workgroup size
00229     std::array<VkSpecializationMapEntry, 2> specEntries{};
00230
00231     specEntries[0].constantID = 0;
00232     specEntries[0].size = sizeof(specializationData.specWorkGroupSizeX);
00233     specEntries[0].offset = 0;
00234
00235     specEntries[1].constantID = 1;

```

```

00236     specEntries[1].size = sizeof(specializationData.specWorkGroupSizeY);
00237     specEntries[1].offset = offsetof(SpecializationData, specWorkGroupSizeY);
00238
00239 // specEntries[2].constantID = 2;
00240 // specEntries[2].size = sizeof(specializationData.specWorkGroupSizeZ);
00241 // specEntries[2].offset = offsetof(SpecializationData, specWorkGroupSizeZ);
00242
00243     VkSpecializationInfo specInfo{};
00244     specInfo.dataSize = sizeof(specializationData);
00245     specInfo.mapEntryCount = static_cast<uint32_t>(specEntries.size());
00246     specInfo.pMapEntries = specEntries.data();
00247     specInfo.pData = &specializationData;
00248
00249     VkPipelineShaderStageCreateInfo compute_shader_integrate_create_info{};
00250     compute_shader_integrate_create_info.sType =
00251         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00252     compute_shader_integrate_create_info.stage = VK_SHADER_STAGE_COMPUTE_BIT;
00253     compute_shader_integrate_create_info.module = pathTracingModule;
00254     compute_shader_integrate_create_info.pSpecializationInfo = &specInfo;
00255     compute_shader_integrate_create_info.pName = "main";
00256
00257 // -- COMPUTE PIPELINE CREATION --
00258     VkComputePipelineCreateInfo compute_pipeline_create_info{};
00259     compute_pipeline_create_info.sType =
00260         VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
00261     compute_pipeline_create_info.stage = compute_shader_integrate_create_info;
00262     compute_pipeline_create_info.layout = pipeline_layout;
00263     compute_pipeline_create_info.flags = 0;
00264 // create compute pipeline
00265     ASSERT_VULKAN(vkCreateComputePipelines(
00266             device->getLogicalDevice(), VK_NULL_HANDLE, 1,
00267             &compute_pipeline_create_info, nullptr, &pipeline),
00268             "Failed to create a compute pipeline!");
00269
00270 // Destroy shader modules, no longer needed after pipeline created
00271     vkDestroyShaderModule(device->getLogicalDevice(), pathTracingModule, nullptr);
00272 }

```

## 6.177 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Src/renderer/PostStage.cpp File Reference

### 6.178 PostStage.cpp

[Go to the documentation of this file.](#)

```

00001 #include "PostStage.hpp"
00002
00003 #include <array>
00004 #include <vector>
00005 #include <filesystem>
00006
00007 #include "File.hpp"
00008 #include "FormatHelper.hpp"
00009 #include "GUI.hpp"
00010 #include "ShaderHelper.hpp"
00011 #include "Vertex.hpp"
00012
00013 #include "VulkanRendererConfig.hpp"
00014
00015 PostStage::PostStage() {}
00016
00017 void PostStage::init(
00018     VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
00019     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00020     this->device = device;
00021     this->vulkanSwapChain = vulkanSwapChain;
00022
00023     createOffscreenTextureSampler();
00024
00025     createPushConstantRange();
00026     createDepthbufferImage();
00027     createRenderpass();
00028     createGraphicsPipeline(descriptorSetLayouts);
00029     createFramebuffer();
00030 }
00031
00032 void PostStage::shaderHotReload(
00033     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {

```

```

00034     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00035     createGraphicsPipeline(descriptor_set_layouts);
00036 }
00037
00038 void PostStage::recordCommands(
00039     VkCommandBuffer& commandBuffer, uint32_t image_index,
00040     const std::vector<VkDescriptorSet>& descriptorSets) {
00041     // information about how to begin a render pass (only needed for graphical
00042     // applications)
00043     VkRenderPassBeginInfo render_pass_begin_info{};
00044     render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
00045     render_pass_begin_info.renderPass = render_pass; // render pass to begin
00046     render_pass_begin_info.renderArea.offset = {
00047         0, 0}; // start point of render pass in pixels
00048     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00049     render_pass_begin_info.renderArea.extent =
00050         swap_chain_extent; // size of region to run render pass on (starting at
00051         // offset)
00052
00053     // make sure the order you put the values into the array matches with the
00054     // attachment order you have defined previous
00055     std::array<VkClearValue, 2> clear_values = {};
00056     clear_values[0].color = {0.2f, 0.65f, 0.4f, 1.0f};
00057     clear_values[1].depthStencil = {1.0f, 0};
00058
00059     render_pass_begin_info.pClearValues = clear_values.data();
00060     render_pass_begin_info.clearValueCount =
00061         static_cast<uint32_t>(clear_values.size());
00062
00063     // used framebuffer depends on the swap chain and therefore is changing for
00064     // each command buffer
00065     render_pass_begin_info.framebuffer = framebuffers[image_index];
00066
00067     // begin render pass
00068     vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
00069         VK_SUBPASS_CONTENTS_INLINE);
00070     auto aspectRatio = static_cast<float>(swap_chain_extent.width) /
00071         static_cast<float>(swap_chain_extent.height);
00072     PushConstantPost pc_post{};
00073     pc_post.aspect_ratio = aspectRatio;
00074     vkCmdPushConstants(commandBuffer, pipeline_layout,
00075         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT,
00076         0, sizeof(PushConstantPost), &pc_post);
00077     vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00078         graphics_pipeline);
00079     vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00080         pipeline_layout, 0,
00081         static_cast<uint32_t>(descriptorSets.size()),
00082         descriptorSets.data(), 0, nullptr);
00083     vkCmdDraw(commandBuffer, 3, 1, 0, 0);
00084
00085     // Rendering gui
00086     ImGui::Render();
00087     ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), commandBuffer);
00088
00089     // end render pass
00090     vkCmdEndRenderPass(commandBuffer);
00091 }
00092
00093 void PostStage::cleanUp() {
00094     depthBufferImage.cleanUp();
00095     for (auto framebuffer : framebuffers) {
00096         vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
00097     }
00098
00099     vkDestroySampler(device->getLogicalDevice(), offscreenTextureSampler,
00100         nullptr);
00101
00102     vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
00103     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00104     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00105 }
00106
00107 PostStage::~PostStage() {}
00108
00109 void PostStage::createDepthbufferImage() {
00110     // get supported format for depth buffer
00111     depth_format = choose_supported_format(
00112         device->getPhysicalDevice(),
00113         {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00114          VK_FORMAT_D24_UNORM_S8_UINT},
00115         VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00116
00117     // create depth buffer image
00118     // MIP LEVELS: for depth texture we only want 1 level :)
00119     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00120     depthBufferImage.createImage(device, swap_chain_extent.width,

```

```

00121                     swap_chain_extent.height, 1, depth_format,
00122                     VK_IMAGE_TILING_OPTIMAL,
00123                     VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
00124                     VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00125
00126         // depth buffer image view
00127         // MIP LEVELS: for depth texture we only want 1 level :)
00128         depthBufferImage.createImageView(device, depth_format,
00129                                         VK_IMAGE_ASPECT_DEPTH_BIT, 1);
00130     }
00131
00132     void PostStage::createOffscreenTextureSampler() {
00133         // sampler create info
00134         VkSamplerCreateInfo sampler_create_info{};
00135         sampler_create_info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
00136         sampler_create_info.magFilter = VK_FILTER_LINEAR;
00137         sampler_create_info.minFilter = VK_FILTER_LINEAR;
00138         sampler_create_info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00139         sampler_create_info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00140         sampler_create_info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00141         sampler_create_info.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
00142         sampler_create_info.unnormalizedCoordinates = VK_FALSE;
00143         sampler_create_info.mipMapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
00144         sampler_create_info.mipLodBias = 0.0f;
00145         sampler_create_info.minLod = 0.0f;
00146         sampler_create_info.maxLod = 0.0f;
00147         sampler_create_info.anisotropyEnable = VK_TRUE;
00148         sampler_create_info.maxAnisotropy = 16; // max anisotropy sample level
00149
00150         VkResult result =
00151             vkCreateSampler(device->getLogicalDevice(), &sampler_create_info, nullptr,
00152                             &offscreenTextureSampler);
00153         ASSERT_VULKAN(result, "Failed to create a texture sampler!")
00154     }
00155
00156     void PostStage::createPushConstantRange() {
00157         push_constant_range.stageFlags =
00158             VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
00159         push_constant_range.offset = 0;
00160         push_constant_range.size = sizeof(PushConstantPost);
00161     }
00162
00163     void PostStage::createRenderpass() {
00164         // Color attachment of render pass
00165         VkAttachmentDescription color_attachment{};
00166         const VkFormat& swap_chain_image_format =
00167             vulkanSwapChain->getSwapChainFormat();
00168         color_attachment.format =
00169             swap_chain_image_format; // format to use for attachment
00170         color_attachment.samples =
00171             VK_SAMPLE_COUNT_1_BIT; // number of samples to write for multisampling
00172         color_attachment.loadOp =
00173             VK_ATTACHMENT_LOAD_OP_CLEAR; // describes what to do with attachment
00174                                         // before rendering
00175         color_attachment.storeOp =
00176             VK_ATTACHMENT_STORE_OP_STORE; // describes what to do with attachment
00177                                         // after rendering
00178         color_attachment.stencilLoadOp =
00179             VK_ATTACHMENT_LOAD_OP_DONT_CARE; // describes what to do with stencil
00180                                         // before rendering
00181         color_attachment.stencilStoreOp =
00182             VK_ATTACHMENT_STORE_OP_DONT_CARE; // describes what to do with stencil
00183                                         // after rendering
00184
00185         // framebuffer data will be stored as an image, but images can be given
00186         // different layouts to give optimal use for certain operations
00187         color_attachment.initialLayout =
00188             VK_IMAGE_LAYOUT_UNDEFINED; // image data layout before render pass starts
00189         color_attachment.finalLayout =
00190             VK_IMAGE_LAYOUT_PRESENT_SRC_KHR; // image data layout after render pass
00191                                         // (to change to)
00192
00193         // depth attachment of render pass
00194         VkAttachmentDescription depth_attachment{};
00195         depth_attachment.format = choose_supported_format(
00196             device->getPhysicalDevice(),
00197             {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00198              VK_FORMAT_D24_UNORM_S8_UINT},
00199             VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00200         depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
00201         depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
00202         depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00203         depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
00204         depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00205         depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
00206         depth_attachment.finalLayout =
00207             VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

```

```

00208 // attachment reference uses an attachment index that refers to index in the
00209 // attachment list passed to renderPassCreateInfo
00210 VkAttachmentReference color_attachment_reference{};
00211 color_attachment_reference.attachment = 0;
00212 color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
00213
00214 // attachment reference
00215 VkAttachmentReference depth_attachment_reference{};
00216 depth_attachment_reference.attachment = 1;
00217 depth_attachment_reference.layout =
00218     VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00219
00220 // information about a particular subpass the render pass is using
00221 VkSubpassDescription subpass{};
00222 subpass.pipelineBindPoint =
00223     VK_PIPELINE_BIND_POINT_GRAPHICS; // pipeline type subpass is to be bound
00224                                         // to
00225 subpass.colorAttachmentCount = 1;
00226 subpass.pColorAttachments = &color_attachment_reference;
00227 subpass.pDepthStencilAttachment = &depth_attachment_reference;
00228
00229 // need to determine when layout transitions occur using subpass dependencies
00230 std::array<VkSubpassDependency, 1> subpass_dependencies;
00231
00232 // conversion from VK_IMAGE_LAYOUT_UNDEFINED to
00233 // VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
00234 subpass_dependencies[0].srcSubpass =
00235     VK_SUBPASS_EXTERNAL; // subpass index (VK_SUBPASS_EXTERNAL = Special
00236                                         // value meaning outside of renderpass)
00237 subpass_dependencies[0].srcStageMask =
00238     VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT; // pipeline stage
00239 subpass_dependencies[0].srcAccessMask =
00240     VK_ACCESS_MEMORY_READ_BIT; // stage access mask (memory access)
00241 subpass_dependencies[0].dstSubpass = 0;
00242 subpass_dependencies[0].dstStageMask =
00243     VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00244 subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
00245                                         VK_ACCESS_COLOR_ATTACHMENT_READ_BIT;
00246 subpass_dependencies[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
00247
00248 std::array<VkAttachmentDescription, 2> render_pass_attachments = {
00249     color_attachment, depth_attachment};
00250
00251 // create info for render pass
00252 VkRenderPassCreateInfo render_pass_create_info{};
00253 render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
00254 render_pass_create_info.attachmentCount =
00255     static_cast<uint32_t>(render_pass_attachments.size());
00256 render_pass_create_info.pAttachments = render_pass_attachments.data();
00257 render_pass_create_info.subpassCount = 1;
00258 render_pass_create_info.pSubpasses = &subpass;
00259 render_pass_create_info.dependencyCount =
00260     static_cast<uint32_t>(subpass_dependencies.size());
00261 render_pass_create_info.pDependencies = subpass_dependencies.data();
00262
00263 VkResult result =
00264     vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
00265                         nullptr, &render_pass);
00266 ASSERT_VULKAN(result, "Failed to create render pass!")
00267
00268 }
00269
00270 void PostStage::createGraphicsPipeline(
00271     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00272     std::stringstream post_shader_dir;
00273     std::filesystem::path cwd = std::filesystem::current_path();
00274     post_shader_dir << cwd.string();
00275     post_shader_dir << RELATIVE_RESOURCE_PATH;
00276     post_shader_dir << "Shaders/post/";
00277
00278     std::string post_vert_shader = "post.vert";
00279     std::string post_frag_shader = "post.frag";
00280
00281     ShaderHelper shaderHelper;
00282     File vertexShaderFile(
00283         shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_vert_shader));
00284     std::vector<char> vertex_shader_code = vertexShaderFile.readCharSequence();
00285     File fragmentShaderFile(
00286         shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_frag_shader));
00287     std::vector<char> fragment_shader_code =
00288         fragmentShaderFile.readCharSequence();
00289
00290     shaderHelper.compileShader(post_shader_dir.str(), post_vert_shader);
00291     shaderHelper.compileShader(post_shader_dir.str(), post_frag_shader);
00292
00293 // build shader modules to link to graphics pipeline
00294 VkShaderModule vertex_shader_module =

```

```
00295     shaderHelper.createShaderModule(device, vertex_shader_code);
00296     VkShaderModule fragment_shader_module =
00297         shaderHelper.createShaderModule(device, fragment_shader_code);
00298
00299     // shader stage creation information
00300     // vertex stage creation information
00301     VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
00302     vertex_shader_create_info.sType =
00303         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00304     vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
00305     vertex_shader_create_info.module = vertex_shader_module;
00306     vertex_shader_create_info.pName = "main";
00307
00308     // fragment stage creation information
00309     VkPipelineShaderStageCreateInfo fragment_shader_create_info{};
00310     fragment_shader_create_info.sType =
00311         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00312     fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
00313     fragment_shader_create_info.module = fragment_shader_module;
00314     fragment_shader_create_info.pName = "main";
00315
00316     std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
00317         vertex_shader_create_info, fragment_shader_create_info};
00318
00319     // how the data for a single vertex (including info such as position, color,
00320     // texture coords, normals, etc) is as a whole
00321     VkVertexInputBindingDescription binding_description{};
00322     binding_description.binding = 0;
00323     binding_description.stride = sizeof(Vertex);
00324     binding_description.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
00325
00326     std::array<VkVertexInputAttributeDescription, 4> attribute_descriptions =
00327         vertex::getVertexInputAttributeDesc();
00328
00329     // CREATE PIPELINE
00330     // 1.) Vertex input
00331     VkPipelineVertexInputStateCreateInfo vertex_input_create_info{};
00332     vertex_input_create_info.sType =
00333         VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
00334     vertex_input_create_info.vertexBindingDescriptionCount = 0;
00335     vertex_input_create_info.pVertexBindingDescriptions = nullptr;
00336     vertex_input_create_info.vertexAttributeDescriptionCount = 0;
00337     vertex_input_create_info.pVertexAttributeDescriptions = nullptr;
00338
00339     // input assembly
00340     VkPipelineInputAssemblyStateCreateInfo input_assembly{};
00341     input_assembly.sType =
00342         VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
00343     input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
00344     input_assembly.primitiveRestartEnable = VK_FALSE;
00345
00346     // viewport & scissor
00347     // create a viewport info struct
00348     VkViewport viewport{};
00349     viewport.x = 0.0f;
00350     viewport.y = 0.0f;
00351     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00352     viewport.width = (float)swap_chain_extent.width;
00353     viewport.height = (float)swap_chain_extent.height;
00354     viewport.minDepth = 0.0f;
00355     viewport.maxDepth = 1.0f;
00356
00357     // create a scissor info struct
00358     VkRect2D scissor{};
00359     scissor.offset = {0, 0};
00360     scissor.extent = swap_chain_extent;
00361
00362     VkPipelineViewportStateCreateInfo viewport_state_create_info{};
00363     viewport_state_create_info.sType =
00364         VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
00365     viewport_state_create_info.viewportCount = 1;
00366     viewport_state_create_info.pViewports = &viewport;
00367     viewport_state_create_info.scissorCount = 1;
00368     viewport_state_create_info.pScissors = &scissor;
00369
00370     // RASTERIZER
00371     VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
00372     rasterizer_create_info.sType =
00373         VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
00374     rasterizer_create_info.depthClampEnable = VK_FALSE;
00375     rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
00376     rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
00377     rasterizer_create_info.lineWidth = 1.0f;
00378     rasterizer_create_info.cullMode = VK_CULL_MODE_NONE;
00379     // winding to determine which side is front; y-coordinate is inverted in
00380     // comparison to OpenGL
00381     rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

```

00382     rasterizer_create_info.depthBiasClamp = VK_FALSE;
00383
00384     // -- MULTISAMPLING --
00385     VkPipelineMultisampleStateCreateInfo multisample_create_info{};
00386     multisample_create_info.sType =
00387         VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
00388     multisample_create_info.sampleShadingEnable = VK_FALSE;
00389     multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
00390
00391     // -- BLENDING --
00392     // blend attachment state
00393     VkPipelineColorBlendAttachmentState color_state{};
00394     color_state.colorWriteMask =
00395         VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
00396         VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
00397
00398     color_state.blendEnable = VK_TRUE;
00399     // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
00400     // (dstColorBlendFactor * old_color)
00401     color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
00402     color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
00403     color_state.colorBlendOp = VK_BLEND_OP_ADD;
00404     color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
00405     color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
00406     color_state.alphaBlendOp = VK_BLEND_OP_ADD;
00407
00408     VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
00409     color_blending_create_info.sType =
00410         VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
00411     color_blending_create_info.logicOpEnable =
00412         VK_FALSE; // alternative to calculations is to use logical operations
00413     color_blending_create_info.logicOp = VK_LOGIC_OP_CLEAR;
00414     color_blending_create_info.attachmentCount = 1;
00415     color_blending_create_info.pAttachments = &color_state;
00416     for (int i = 0; i < 4; i++) {
00417         color_blending_create_info.blendConstants[0] = 0.f;
00418     }
00419     // -- PIPELINE LAYOUT --
00420     VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00421     pipeline_layout_create_info.sType =
00422         VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00423     pipeline_layout_create_info.setLayoutCount =
00424         static_cast<uint32_t>(descriptorSetLayouts.size());
00425     pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00426     pipeline_layout_create_info.pushConstantRangeCount = 1;
00427     pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;
00428
00429     // create pipeline layout
00430     VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00431                                             &pipeline_layout_create_info,
00432                                             nullptr, &pipeline_layout);
00433     ASSERT_VULKAN(result, "Failed to create pipeline layout!")
00434
00435     // -- DEPTH STENCIL TESTING --
00436     VkPipelineDepthStencilStateCreateInfo depth_stencil_create_info{};
00437     depth_stencil_create_info.sType =
00438         VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
00439     depth_stencil_create_info.depthTestEnable = VK_TRUE;
00440     depth_stencil_create_info.depthWriteEnable = VK_TRUE;
00441     depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
00442     depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
00443     depth_stencil_create_info.stencilTestEnable = VK_FALSE;
00444
00445     // -- GRAPHICS PIPELINE CREATION --
00446     VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
00447     graphics_pipeline_create_info.sType =
00448         VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
00449     graphics_pipeline_create_info.stageCount =
00450         static_cast<uint32_t>(shader_stages.size());
00451     graphics_pipeline_create_info.pStages = shader_stages.data();
00452     graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
00453     graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
00454     graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
00455     graphics_pipeline_create_info.pDynamicState = nullptr;
00456     graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
00457     graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
00458     graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
00459     graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
00460     graphics_pipeline_create_info.layout = pipeline_layout;
00461     graphics_pipeline_create_info.renderPass = render_pass;
00462     graphics_pipeline_create_info.subpass = 0;
00463
00464     // pipeline derivatives : can create multiple pipelines that derive from one
00465     // another for optimization
00466     graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
00467     graphics_pipeline_create_info.basePipelineIndex = -1;
00468

```

```

00469 // create graphics pipeline
00470 result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
00471                                     1, &graphics_pipeline_create_info, nullptr,
00472                                     &graphics_pipeline);
00473 ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")
00474
00475 // Destroy shader modules, no longer needed after pipeline created
00476 vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
00477                         nullptr);
00478 vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,
00479                         nullptr);
00480 }
00481
00482 void PostStage::createFramebuffer() {
00483     // resize framebuffer size to equal swap chain image count
00484     framebuffers.resize(vulkanSwapChain->getNumberSwapChainImages());
00485
00486     for (size_t i = 0; i < vulkanSwapChain->getNumberSwapChainImages(); i++) {
00487         Texture& swap_chain_image = vulkanSwapChain->getSwapChainImage(i);
00488
00489         std::array<VkImageView, 2> attachments = {swap_chain_image.getImageView(),
00490                                                 depthBufferImage.getImageView()};
00491
00492         VkFramebufferCreateInfo frame_buffer_create_info{};
00493         frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
00494         frame_buffer_create_info.renderPass =
00495             render_pass; // render pass layout the framebuffer will be used with
00496         frame_buffer_create_info.attachmentCount =
00497             static_cast<uint32_t>(attachments.size());
00498         frame_buffer_create_info.pAttachments =
00499             attachments.data(); // list of attachments (1:1 with render pass)
00500         const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00501         frame_buffer_create_info.width =
00502             swap_chain_extent.width; // framebuffer width
00503         frame_buffer_create_info.height =
00504             swap_chain_extent.height; // framebuffer height
00505         frame_buffer_create_info.layers = 1; // framebuffer layer
00506
00507         VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
00508                                             &frame_buffer_create_info, nullptr,
00509                                             &framebuffers[i]);
00510         ASSERT_VULKAN(result, "Failed to create framebuffer!")
00511     }
00512 }

```

## 6.179 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Src/renderer/Rasterizer.cpp File Reference

### 6.180 Rasterizer.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Rasterizer.hpp"
00002
00003 #include <array>
00004 #include <vector>
00005 #include <filesystem>
00006
00007 #include "File.hpp"
00008 #include "FormatHelper.hpp"
00009 #include "ShaderHelper.hpp"
00010 #include "Vertex.hpp"
00011
00012 #include "VulkanRendererConfig.hpp"
00013
00014 Rasterizer::Rasterizer() {}
00015
00016 void Rasterizer::init(
00017     VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
00018     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts,
00019     VkCommandPool& commandPool) {
00020     this->device = device;
00021     this->vulkanSwapChain = vulkanSwapChain;
00022
00023     createTextures(commandPool);
00024     createRenderPass();
00025     createPushConstantRange();
00026     createGraphicsPipeline(descriptorSetLayouts);

```

```

00027     createFramebuffer();
00028 }
00029
00030 void Rasterizer::shaderHotReload(
00031     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
00032     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00033     createGraphicsPipeline(descriptor_set_layouts);
00034 }
00035
00036 Texture& Rasterizer::getOffscreenTexture(uint32_t index) {
00037     return offscreenTextures[index];
00038 }
00039
00040 void Rasterizer::setPushConstant(PushConstantRasterizer pushConstant) {
00041     this->pushConstant = pushConstant;
00042 }
00043
00044 void Rasterizer::recordCommands(
00045     VkCommandBuffer& commandBuffer, uint32_t image_index, Scene* scene,
00046     const std::vector<VkDescriptorSet>& descriptorSets) {
00047     // information about how to begin a render pass (only needed for graphical
00048     // applications)
00049     VkRenderPassBeginInfo render_pass_begin_info{};
00050     render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
00051     render_pass_begin_info.renderPass = render_pass;
00052     render_pass_begin_info.renderArea.offset = {0, 0};
00053     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00054     render_pass_begin_info.renderArea.extent = swap_chain_extent;
00055
00056     // make sure the order you put the values into the array matches with the
00057     // attachment order you have defined previous
00058     std::array<VkClearValue, 2> clear_values = {};
00059     clear_values[0].color = {0.2f, 0.65f, 0.4f, 1.0f};
00060     clear_values[1].depthStencil = {1.0f, 0};
00061
00062     render_pass_begin_info.pClearValues = clear_values.data();
00063     render_pass_begin_info.clearValueCount =
00064         static_cast<uint32_t>(clear_values.size());
00065     render_pass_begin_info.framebuffer = framebuffer[image_index];
00066
00067     // begin render pass
00068     vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
00069                           VK_SUBPASS_CONTENTS_INLINE);
00070
00071     // bind pipeline to be used in render pass
00072     vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00073                        graphics_pipeline);
00074
00075     for (uint32_t m = 0; m < static_cast<uint32_t>(scene->getModelCount()); m++) {
00076         // for GCC doesn't allow references on rvalues go like that ...
00077         pushConstant.model = scene->getModelMatrix(0);
00078         // just "Push" constants to given shader stage directly (no buffer)
00079         vkCmdPushConstants(
00080             commandBuffer, pipeline_layout,
00081             VK_SHADER_STAGE_VERTEX_BIT,           // stage to push constants to
00082             0,                                // offset to push constants to update
00083             sizeof(PushConstantRasterizer),      // size of data being pushed
00084             &pushConstant); // using model of current mesh (can be array)
00085
00086     for (unsigned int k = 0; k < scene->getMeshCount(m); k++) {
00087         // list of vertex buffers we want to draw
00088         VkBuffer vertex_buffers[] = {
00089             scene->getVertexBuffer(m, k)}; // buffers to bind
00090         VkDeviceSize offsets[] = {0};
00091         vkCmdBindVertexBuffers(
00092             commandBuffer, 0, 1, vertex_buffers,
00093             offsets); // command to bind vertex buffer before drawing with them
00094
00095         // bind mesh index buffer with 0 offset and using the uint32 type
00096         vkCmdBindIndexBuffer(commandBuffer, scene->getIndexBuffer(m, k), 0,
00097                               VK_INDEX_TYPE_UINT32);
00098
00099         // bind descriptor sets
00100         vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00101                                 pipeline_layout, 0,
00102                                 static_cast<uint32_t>(descriptorSets.size()),
00103                                 descriptorSets.data(), 0, nullptr);
00104
00105         // execute pipeline
00106         vkCmdDrawIndexed(commandBuffer,
00107                           static_cast<uint32_t>(scene->getIndexCount(m, k)), 1, 0,
00108                           0, 0);
00109     }
00110 }
00111
00112     // end render pass
00113     vkCmdEndRenderPass(commandBuffer);

```

```
0014 }
0015
0016 void Rasterizer::cleanUp() {
0017     for (auto framebuffer : framebuffer) {
0018         vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
0019     }
0020
0021     for (Texture texture : offscreenTextures) {
0022         texture.cleanUp();
0023     }
0024
0025     depthBufferImage.cleanUp();
0026
0027     vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
0028     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
0029     vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
0030 }
0031
0032 Rasterizer::~Rasterizer() {}
0033
0034 void Rasterizer::createRenderPass() {
0035     // Color attachment of render pass
0036     VkAttachmentDescription color_attachment{};
0037     const VkFormat& swap_chain_image_format =
0038         vulkanSwapChain->getSwapChainFormat();
0039     color_attachment.format =
0040         swap_chain_image_format; // format to use for attachment
0041     color_attachment.samples =
0042         VK_SAMPLE_COUNT_1_BIT; // number of samples to write for multisampling
0043     color_attachment.loadOp =
0044         VK_ATTACHMENT_LOAD_OP_CLEAR; // describes what to do with attachment
0045                         // before rendering
0046     color_attachment.storeOp =
0047         VK_ATTACHMENT_STORE_OP_STORE; // describes what to do with attachment
0048                         // after rendering
0049     color_attachment.stencilLoadOp =
0050         VK_ATTACHMENT_LOAD_OP_DONT_CARE; // describes what to do with stencil
0051                         // before rendering
0052     color_attachment.stencilStoreOp =
0053         VK_ATTACHMENT_STORE_OP_DONT_CARE; // describes what to do with stencil
0054                         // after rendering
0055
0056     // framebuffer data will be stored as an image, but images can be given
0057     // different layouts to give optimal use for certain operations
0058     color_attachment.initialLayout =
0059         VK_IMAGE_LAYOUT_GENERAL; // image data layout before render pass starts
0060     color_attachment.finalLayout =
0061         VK_IMAGE_LAYOUT_GENERAL; // image data layout after render pass (to
0062                         // change to)
0063
0064     // depth attachment of render pass
0065     VkAttachmentDescription depth_attachment{};
0066     depth_attachment.format = choose_supported_format(
0067         device->getPhysicalDevice(),
0068         {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
0069          VK_FORMAT_D24_UNORM_S8_UINT},
0070         VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
0071
0072     depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
0073     depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
0074     depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
0075     depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
0076     depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
0077     depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
0078     depth_attachment.finalLayout =
0079         VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
0080
0081     // attachment reference uses an attachment index that refers to index in the
0082     // attachment list passed to renderPassCreateInfo
0083     VkAttachmentReference color_attachment_reference{};
0084     color_attachment_reference.attachment = 0;
0085     color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
0086
0087     // attachment reference
0088     VkAttachmentReference depth_attachment_reference{};
0089     depth_attachment_reference.attachment = 1;
0090     depth_attachment_reference.layout =
0091         VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
0092
0093     // information about a particular subpass the render pass is using
0094     VkSubpassDescription subpass{};
0095     subpass.pipelineBindPoint =
0096         VK_PIPELINE_BIND_POINT_GRAPHICS; // pipeline type subpass is to be bound
0097                         // to
0098     subpass.colorAttachmentCount = 1;
0099     subpass.pColorAttachments = &color_attachment_reference;
0100     subpass.pDepthStencilAttachment = &depth_attachment_reference;
```

```

00201
00202     // need to determine when layout transitions occur using subpass dependencies
00203     std::array<VkSubpassDependency, 1> subpass_dependencies;
00204
00205     // conversion from VK_IMAGE_LAYOUT_UNDEFINED to
00206     // VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
00207     subpass_dependencies[0].srcSubpass =
00208         VK_SUBPASS_EXTERNAL; // subpass index (VK_SUBPASS_EXTERNAL = Special
00209         // value meaning outside of renderpass)
00210     subpass_dependencies[0].srcStageMask =
00211         VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT; // pipeline stage
00212     subpass_dependencies[0].srcAccessMask =
00213         0; // stage access mask (memory access)
00214
00215     // but must happen before ...
00216     subpass_dependencies[0].dstSubpass = 0;
00217     subpass_dependencies[0].dstStageMask =
00218         VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00219     subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
00220     subpass_dependencies[0].dependencyFlags = 0; // VK_DEPENDENCY_BY_REGION_BIT;
00221
00222     std::array<VkAttachmentDescription, 2> render_pass_attachments = {
00223         color_attachment, depth_attachment};
00224
00225     // create info for render pass
00226     VkRenderPassCreateInfo render_pass_create_info{};
00227     render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
00228     render_pass_create_info.attachmentCount =
00229         static_cast<uint32_t>(render_pass_attachments.size());
00230     render_pass_create_info.pAttachments = render_pass_attachments.data();
00231     render_pass_create_info.subpassCount = 1;
00232     render_pass_create_info.pSubpasses = &subpass;
00233     render_pass_create_info.dependencyCount =
00234         static_cast<uint32_t>(subpass_dependencies.size());
00235     render_pass_create_info.pDependencies = subpass_dependencies.data();
00236
00237     VkResult result =
00238         vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
00239                             nullptr, &render_pass);
00240     ASSERT_VULKAN(result, "Failed to create render pass!")
00241 }
00242
00243 void Rasterizer::createFramebuffer() {
00244     framebuffer.resize(vulkanSwapChain->getNumberSwapChainImages());
00245
00246     for (size_t i = 0; i < framebuffer.size(); i++) {
00247         std::array<VkImageView, 2> attachments = {
00248             offscreenTextures[i].getImageView(), depthBufferImage.getImageView()};
00249
00250         VkFramebufferCreateInfo frame_buffer_create_info{};
00251         frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
00252         frame_buffer_create_info.renderPass = render_pass;
00253         frame_buffer_create_info.attachmentCount =
00254             static_cast<uint32_t>(attachments.size());
00255         frame_buffer_create_info.pAttachments = attachments.data();
00256         const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00257         frame_buffer_create_info.width = swap_chain_extent.width;
00258         frame_buffer_create_info.height = swap_chain_extent.height;
00259         frame_buffer_create_info.layers = 1;
00260
00261         VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
00262                                             &frame_buffer_create_info, nullptr,
00263                                             &framebuffer[i]);
00264         ASSERT_VULKAN(result, "Failed to create framebuffer!");
00265     }
00266 }
00267
00268 void Rasterizer::createPushConstantRange() {
00269     // define push constant values (no 'create' needed)
00270     push_constant_range.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
00271     push_constant_range.offset = 0;
00272     push_constant_range.size = sizeof(PushConstantRasterizer);
00273 }
00274
00275 void Rasterizer::createTextures(VkCommandPool& commandPool) {
00276     offscreenTextures.resize(vulkanSwapChain->getNumberSwapChainImages());
00277
00278     VkCommandBuffer cmdBuffer = commandBufferManager.beginCommandBuffer(
00279         device->getLogicalDevice(), commandPool);
00280
00281     for (uint32_t index = 0;
00282         index <
00283             static_cast<uint32_t>(vulkanSwapChain->getNumberSwapChainImages());
00284         index++) {
00285         Texture texture{};
00286         const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00287         const VkFormat& swap_chain_image_format =

```

```
00288     vulkanSwapChain->getSwapChainFormat();
00289
00290     texture.createImage(
00291         device, swap_chain_extent.width, swap_chain_extent.height, 1,
00292         swap_chain_image_format, VK_IMAGE_TILING_OPTIMAL,
00293         VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT |
00294             VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT,
00295         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00296
00297     texture.createImageView(device, swap_chain_image_format,
00298                             VK_IMAGE_ASPECT_COLOR_BIT, 1);
00299
00300     // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
00301     VulkanImage& image = texture.getVulkanImage();
00302     image.transitionImageLayout(cmdBuffer, VK_IMAGE_LAYOUT_UNDEFINED,
00303                                 VK_IMAGE_LAYOUT_GENERAL, 1,
00304                                 VK_IMAGE_ASPECT_COLOR_BIT);
00305
00306     offscreenTextures[index] = texture;
00307 }
00308
00309 VkFormat depth_format = choose_supported_format(
00310     device->getPhysicalDevice(),
00311     {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00312      VK_FORMAT_D24_UNORM_S8_UINT},
00313     VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00314
00315 // create depth buffer image
00316 // MIP LEVELS: for depth texture we only want 1 level :)
00317 const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00318 depthBufferImage.createImage(device, swap_chain_extent.width,
00319                             swap_chain_extent.height, 1, depth_format,
00320                             VK_IMAGE_TILING_OPTIMAL,
00321                             VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
00322                             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00323
00324 // depth buffer image view
00325 // MIP LEVELS: for depth texture we only want 1 level :)
00326 depthBufferImage.createImageView(
00327     device, depth_format,
00328     VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);
00329
00330 // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
00331 VulkanImage& vulkanImage = depthBufferImage.getVulkanImage();
00332 vulkanImage.transitionImageLayout(
00333     device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
00334     VK_IMAGE_LAYOUT_UNDEFINED,
00335     VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
00336     VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);
00337
00338 commandBufferManager.endAndSubmitCommandBuffer(
00339     device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
00340     cmdBuffer);
00341 }
00342
00343 void Rasterizer::createGraphicsPipeline(
00344     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00345     std::stringstream rasterizer_shader_dir;
00346     std::filesystem::path cwd = std::filesystem::current_path();
00347     rasterizer_shader_dir << cwd.string();
00348     rasterizer_shader_dir << RELATIVE_RESOURCE_PATH;
00349     rasterizer_shader_dir << "Shaders/rasterizer/";
00350
00351     ShaderHelper shaderHelper;
00352     shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.vert");
00353     shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.frag");
00354
00355     File vertexFile(
00356         shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.vert"));
00357     File fragmentFile(
00358         shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.frag"));
00359     std::vector<char> vertex_shader_code = vertexFile.readCharSequence();
00360     std::vector<char> fragment_shader_code = fragmentFile.readCharSequence();
00361
00362     // build shader modules to link to graphics pipeline
00363     VkShaderModule vertex_shader_module =
00364         shaderHelper.createShaderModule(device, vertex_shader_code);
00365     VkShaderModule fragment_shader_module =
00366         shaderHelper.createShaderModule(device, fragment_shader_code);
00367
00368     // shader stage creation information
00369     // vertex stage creation information
00370     VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
00371     vertex_shader_create_info.sType =
00372         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00373     vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
00374     vertex_shader_create_info.module = vertex_shader_module;
```

```

00375     vertex_shader_create_info.pName = "main";
00376
00377     // fragment stage creation information
00378     VKPipelineShaderStageCreateInfo fragment_shader_create_info{};
00379     fragment_shader_create_info.sType =
00380         VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00381     fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
00382     fragment_shader_create_info.module = fragment_shader_module;
00383     fragment_shader_create_info.pName = "main";
00384
00385     std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
00386         vertex_shader_create_info, fragment_shader_create_info};
00387
00388     // how the data for a single vertex (including info such as position, color,
00389     // texture coords, normals, etc) is as a whole
00390     VkVertexInputBindingDescription binding_description{};
00391     binding_description.binding = 0;
00392     binding_description.stride = sizeof(Vertex);
00393     binding_description.inputRate =
00394         VK_VERTEX_INPUT_RATE_VERTEX; // how to move between data after each
00395         // vertex.
00396
00397     // how the data for an attribute is defined within a vertex
00398     std::array<VkVertexInputAttributeDescription, 4> attribute_descriptions =
00399         vertex::getVertexInputAttributeDesc();
00400
00401     // CREATE PIPELINE
00402     // 1.) Vertex input
00403     VKPipelineVertexInputStateCreateInfo vertex_input_create_info{};
00404     vertex_input_create_info.sType =
00405         VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
00406     vertex_input_create_info.vertexBindingDescriptionCount = 1;
00407     vertex_input_create_info.pVertexBindingDescriptions = &binding_description;
00408     vertex_input_create_info.vertexAttributeDescriptionCount =
00409         static_cast<uint32_t>(attribute_descriptions.size());
00410     vertex_input_create_info.pVertexAttributeDescriptions =
00411         attribute_descriptions.data();
00412
00413     // input assembly
00414     VkPipelineInputAssemblyStateCreateInfo input_assembly{};
00415     input_assembly.sType =
00416         VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
00417     input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
00418     input_assembly.primitiveRestartEnable = VK_FALSE;
00419
00420     // viewport & scissor
00421     // create a viewport info struct
00422     VkViewport viewport{};
00423     viewport.x = 0.0f;
00424     viewport.y = 0.0f;
00425     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00426     viewport.width = (float)swap_chain_extent.width;
00427     viewport.height = (float)swap_chain_extent.height;
00428     viewport.minDepth = 0.0f;
00429     viewport.maxDepth = 1.0f;
00430
00431     // create a scissor info struct
00432     VkRect2D scissor{};
00433     scissor.offset = {0, 0};
00434     scissor.extent = swap_chain_extent;
00435
00436     VkPipelineViewportStateCreateInfo viewport_state_create_info{};
00437     viewport_state_create_info.sType =
00438         VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
00439     viewport_state_create_info.viewportCount = 1;
00440     viewport_state_create_info.pViewports = &viewport;
00441     viewport_state_create_info.scissorCount = 1;
00442     viewport_state_create_info.pScissors = &scissor;
00443
00444     // RASTERIZER
00445     VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
00446     rasterizer_create_info.sType =
00447         VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
00448     rasterizer_create_info.info.depthClampEnable = VK_FALSE;
00449     rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
00450     rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
00451     rasterizer_create_info.lineWidth = 1.0f;
00452     rasterizer_create_info.cullMode = VK_CULL_MODE_BACK_BIT; //
00453     // winding to determine which side is front; y-coordinate is inverted in
00454     // comparison to OpenGL
00455     rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
00456     rasterizer_create_info.depthBiasClamp = VK_FALSE;
00457
00458     // -- MULTISAMPLING --
00459     VkPipelineMultisampleStateCreateInfo multisample_create_info{};
00460     multisample_create_info.sType =
00461         VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;

```

```

00462     multisample_create_info.sampleShadingEnable = VK_FALSE;
00463     multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
00464
00465     // -- BLENDING --
00466     // blend attachment state
00467     VkPipelineColorBlendAttachmentState color_state{};
00468     color_state.colorWriteMask =
00469         VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
00470         VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
00471
00472     color_state.blendEnable = VK_TRUE;
00473     // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
00474     // (dstColorBlendFactor * old_color)
00475     color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
00476     color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
00477     color_state.colorBlendOp = VK_BLEND_OP_ADD;
00478     color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
00479     color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
00480     color_state.alphaBlendOp = VK_BLEND_OP_ADD;
00481
00482     VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
00483     color_blending_create_info.sType =
00484         VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
00485     color_blending_create_info.logicOpEnable = VK_FALSE;
00486     color_blending_create_info.attachmentCount = 1;
00487     color_blending_create_info.pAttachments = &color_state;
00488
00489     // -- PIPELINE LAYOUT --
00490     VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00491     pipeline_layout_create_info.sType =
00492         VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00493     pipeline_layout_create_info.setLayoutCount =
00494         static_cast<uint32_t>(descriptorSetLayouts.size());
00495     pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00496     pipeline_layout_create_info.pushConstantRangeCount = 1;
00497     pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;
00498
00499     // create pipeline layout
00500     VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00501                                             &pipeline_layout_create_info,
00502                                             nullptr, &pipeline_layout);
00503     ASSERT_VULKAN(result, "Failed to create pipeline layout!")
00504
00505     // -- DEPTH_STENCIL TESTING --
00506     VkPipelineDepthStencilCreateInfo depth_stencil_create_info{};
00507     depth_stencil_create_info.sType =
00508         VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
00509     depth_stencil_create_info.depthTestEnable = VK_TRUE;
00510     depth_stencil_create_info.depthWriteEnable = VK_TRUE;
00511     depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS;
00512     depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
00513     depth_stencil_create_info.stencilTestEnable = VK_FALSE;
00514
00515     // -- GRAPHICS PIPELINE CREATION --
00516     VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
00517     graphics_pipeline_create_info.sType =
00518         VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
00519     graphics_pipeline_create_info.stageCount =
00520         static_cast<uint32_t>(shader_stages.size());
00521     graphics_pipeline_create_info.pStages = shader_stages.data();
00522     graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
00523     graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
00524     graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
00525     graphics_pipeline_create_info.pDynamicState = nullptr;
00526     graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
00527     graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
00528     graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
00529     graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
00530     graphics_pipeline_create_info.layout = pipeline_layout;
00531     graphics_pipeline_create_info.renderPass = render_pass;
00532     graphics_pipeline_create_info.subpass = 0;
00533
00534     // pipeline derivatives : can create multiple pipelines that derive from one
00535     // another for optimization
00536     graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
00537     graphics_pipeline_create_info.basePipelineIndex = -1;
00538
00539     // create graphics pipeline
00540     result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
00541                                         1, &graphics_pipeline_create_info, nullptr,
00542                                         &graphics_pipeline);
00543     ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")
00544
00545     // Destroy shader modules, no longer needed after pipeline created
00546     vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
00547                           nullptr);
00548     vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,

```

```
00549             nullptr);
00550 }
```

## 6.181 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/← Src/renderer/Raytracing.cpp File Reference

### 6.182 Raytracing.cpp

[Go to the documentation of this file.](#)

```
00001 #include "Raytracing.hpp"
00002
00003 #include <array>
00004 #include <vector>
00005 #include <filesystem>
00006
00007 #include "File.hpp"
00008 #include "MemoryHelper.hpp"
00009 #include "ShaderHelper.hpp"
00010 #include "VulkanRendererConfig.hpp"
00011
00012 Raytracing::Raytracing() {}
00013
00014 void Raytracing::init(
00015     VulkanDevice* device,
00016     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00017     this->device = device;
00018
00019     createPCRange();
00020     createGraphicsPipeline(descriptorSetLayouts);
00021     createSBT();
00022 }
00023
00024 void Raytracing::shaderHotReload(
00025     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
00026     vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
00027     createGraphicsPipeline(descriptor_set_layouts);
00028 }
00029
00030 void Raytracing::recordCommands(
00031     VkCommandBuffer& commandBuffer, VulkanSwapChain* vulkanSwapChain,
00032     const std::vector<VkDescriptorSet>& descriptorSets) {
00033     uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
00034     uint32_t handle_size_aligned =
00035         align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);
00036
00037     PFN_vkGetBufferDeviceAddressKHR vkGetBufferDeviceAddressKHR =
00038         reinterpret_cast<PFN_vkGetBufferDeviceAddressKHR>(vkGetDeviceProcAddr(
00039             device->getLogicalDevice(), "vkGetBufferDeviceAddressKHR"));
00040
00041     PFN_vkCmdTraceRaysKHR pvkCmdTraceRaysKHR =
00042         (PFN_vkCmdTraceRaysKHR)vkGetDeviceProcAddr(device->getLogicalDevice(),
00043             "vkCmdTraceRaysKHR");
00044
00045     VkBufferDeviceAddressInfoKHR bufferDeviceAI{};
00046     bufferDeviceAI.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00047     bufferDeviceAI.buffer = raygenShaderBindingTableBuffer.getBuffer();
00048
00049     rgen_region.deviceAddress =
00050         vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00051     rgen_region.stride = handle_size_aligned;
00052     rgen_region.size = handle_size_aligned;
00053
00054     bufferDeviceAI.buffer = missShaderBindingTableBuffer.getBuffer();
00055     miss_region.deviceAddress =
00056         vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00057     miss_region.stride = handle_size_aligned;
00058     miss_region.size = handle_size_aligned;
00059
00060     bufferDeviceAI.buffer = hitShaderBindingTableBuffer.getBuffer();
00061     hit_region.deviceAddress =
00062         vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00063     hit_region.stride = handle_size_aligned;
00064     hit_region.size = handle_size_aligned;
00065
00066 // for GCC doen't allow references on rvalues go like that ...
00067 pc.clear_color = {0.2f, 0.65f, 0.4f, 1.0f};
00068 // just "Push" constants to given shader stage directly (no buffer)
```

```

00069     vkCmdPushConstants(commandBuffer, pipeline_layout,
00070                           VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00071                               VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00072                               VK_SHADER_STAGE_MISS_BIT_KHR,
00073                           0, sizeof(PushConstantRaytracing), &pc);
00074
00075     vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
00076                         graphicsPipeline);
00077
00078     vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
00079                             pipeline_layout, 0,
00080                             static_cast<uint32_t>(descriptorSets.size()),
00081                             descriptorSets.data(), 0, nullptr);
00082
00083     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00084     pvkCmdTraceRaysKHR(commandBuffer, &rgen_region, &miss_region, &hit_region,
00085                          &call_region, swap_chain_extent.width,
00086                          swap_chain_extent.height, 1);
00087 }
00088
00089 void Raytracing::cleanUp() {
00090     shaderBindingTableBuffer.cleanUp();
00091     raygenShaderBindingTableBuffer.cleanUp();
00092     missShaderBindingTableBuffer.cleanUp();
00093     hitShaderBindingTableBuffer.cleanUp();
00094
00095     vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
00096     vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00097 }
00098
00099 Raytracing::~Raytracing() {}
00100
00101 void Raytracing::createPCRange() {
00102     // define push constant values (no 'create' needed)
00103     pc_ranges.stageFlags = VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00104                               VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00105                               VK_SHADER_STAGE_MISS_BIT_KHR;
00106     pc_ranges.offset = 0;
00107     pc_ranges.size = sizeof(PushConstantRaytracing); // size of data being passed
00108 }
00109
00110 void Raytracing::createGraphicsPipeline(
00111     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00112     PFN_vkCreateRayTracingPipelinesKHR pvkCreateRayTracingPipelinesKHR =
00113         (PFN_vkCreateRayTracingPipelinesKHR)vkGetDeviceProcAddr(
00114             device->getLogicalDevice(), "vkCreateRayTracingPipelinesKHR");
00115
00116     std::stringstream raytracing_shader_dir;
00117     std::filesystem::path cwd = std::filesystem::current_path();
00118     raytracing_shader_dir << cwd.string();
00119     raytracing_shader_dir << RELATIVE_RESOURCE_PATH;
00120     raytracing_shader_dir << "Shaders/raytracing/";
00121
00122     std::string raygen_shader = "raytrace.rgen";
00123     std::string chit_shader = "raytrace.rchit";
00124     std::string miss_shader = "raytrace.rmiss";
00125     std::string shadow_shader = "shadow.rmiss";
00126
00127     ShaderHelper shaderHelper;
00128     shaderHelper.compileShader(raytracing_shader_dir.str(), raygen_shader);
00129     shaderHelper.compileShader(raytracing_shader_dir.str(), chit_shader);
00130     shaderHelper.compileShader(raytracing_shader_dir.str(), miss_shader);
00131     shaderHelper.compileShader(raytracing_shader_dir.str(), shadow_shader);
00132
00133     File raygenFile(
00134         shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), raygen_shader));
00135     File raychitFile(
00136         shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), chit_shader));
00137     File raymissFile(
00138         shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), miss_shader));
00139     File shadowFile(
00140         shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), shadow_shader));
00141
00142     std::vector<char> raygen_shader_code = raygenFile.readCharSequence();
00143     std::vector<char> raychit_shader_code = raychitFile.readCharSequence();
00144     std::vector<char> raymiss_shader_code = raymissFile.readCharSequence();
00145     std::vector<char> shadow_shader_code = shadowFile.readCharSequence();
00146
00147     // build shader modules to link to graphics pipeline
00148     VkShaderModule raygen_shader_module =
00149         shaderHelper.createShaderModule(device, raygen_shader_code);
00150     VkShaderModule raychit_shader_module =
00151         shaderHelper.createShaderModule(device, raychit_shader_code);
00152     VkShaderModule raymiss_shader_module =
00153         shaderHelper.createShaderModule(device, raymiss_shader_code);
00154     VkShaderModule shadow_shader_module =
00155         shaderHelper.createShaderModule(device, shadow_shader_code);

```

```

00156
00157 // create all shader stage infos for creating a group
00158 VkPipelineShaderStageCreateInfo rgen_shader_stage_info{};
00159 rgen_shader_stage_info.sType =
00160     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00161 rgen_shader_stage_info.stage = VK_SHADER_STAGE_RAYGEN_BIT_KHR;
00162 rgen_shader_stage_info.module = raygen_shader_module;
00163 rgen_shader_stage_info.pName = "main";
00164
00165 VkPipelineShaderStageCreateInfo rmiss_shader_stage_info{};
00166 rmiss_shader_stage_info.sType =
00167     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00168 rmiss_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
00169 rmiss_shader_stage_info.module = raymiss_shader_module;
00170 rmiss_shader_stage_info.pName = "main";
00171
00172 VkPipelineShaderStageCreateInfo shadow_shader_stage_info{};
00173 shadow_shader_stage_info.sType =
00174     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00175 shadow_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
00176 shadow_shader_stage_info.module = shadow_shader_module;
00177 shadow_shader_stage_info.pName = "main";
00178
00179 VkPipelineShaderStageCreateInfo rchit_shader_stage_info{};
00180 rchit_shader_stage_info.sType =
00181     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00182 rchit_shader_stage_info.stage = VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR;
00183 rchit_shader_stage_info.module = rayhit_shader_module;
00184 rchit_shader_stage_info.pName = "main";
00185
00186 // we have all shader stages together
00187 std::array<VkPipelineShaderStageCreateInfo, 4> shader_stages = {
00188     rgen_shader_stage_info, rmiss_shader_stage_info, shadow_shader_stage_info,
00189     rchit_shader_stage_info};
00190
00191 enum StageIndices { eRaygen, eMiss, eMiss2, eClosestHit, eShaderGroupCount };
00192
00193 shader_groups.reserve(4);
00194 VkRayTracingShaderGroupCreateInfoKHR shader_group_create_infos[4];
00195
00196 shader_group_create_infos[0].sType =
00197     VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00198 shader_group_create_infos[0].pNext = nullptr;
00199 shader_group_create_infos[0].type =
00200     VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00201 shader_group_create_infos[0].generalShader = eRaygen;
00202 shader_group_create_infos[0].closestHitShader = VK_SHADER_UNUSED_KHR;
00203 shader_group_create_infos[0].anyHitShader = VK_SHADER_UNUSED_KHR;
00204 shader_group_create_infos[0].intersectionShader = VK_SHADER_UNUSED_KHR;
00205 shader_group_create_infos[0].pShaderGroupCaptureReplayHandle = nullptr;
00206
00207 shader_groups.push_back(shader_group_create_infos[0]);
00208
00209 shader_group_create_infos[1].sType =
00210     VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00211 shader_group_create_infos[1].pNext = nullptr;
00212 shader_group_create_infos[1].type =
00213     VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00214 shader_group_create_infos[1].generalShader = eMiss;
00215 shader_group_create_infos[1].closestHitShader = VK_SHADER_UNUSED_KHR;
00216 shader_group_create_infos[1].anyHitShader = VK_SHADER_UNUSED_KHR;
00217 shader_group_create_infos[1].intersectionShader = VK_SHADER_UNUSED_KHR;
00218 shader_group_create_infos[1].pShaderGroupCaptureReplayHandle = nullptr;
00219
00220 shader_groups.push_back(shader_group_create_infos[1]);
00221
00222 shader_group_create_infos[2].sType =
00223     VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00224 shader_group_create_infos[2].pNext = nullptr;
00225 shader_group_create_infos[2].type =
00226     VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00227 shader_group_create_infos[2].generalShader = eMiss2;
00228 shader_group_create_infos[2].closestHitShader = VK_SHADER_UNUSED_KHR;
00229 shader_group_create_infos[2].anyHitShader = VK_SHADER_UNUSED_KHR;
00230 shader_group_create_infos[2].intersectionShader = VK_SHADER_UNUSED_KHR;
00231 shader_group_create_infos[2].pShaderGroupCaptureReplayHandle = nullptr;
00232
00233 shader_groups.push_back(shader_group_create_infos[2]);
00234
00235 shader_group_create_infos[3].sType =
00236     VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00237 shader_group_create_infos[3].pNext = nullptr;
00238 shader_group_create_infos[3].type =
00239     VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR;
00240 shader_group_create_infos[3].generalShader = VK_SHADER_UNUSED_KHR;
00241 shader_group_create_infos[3].closestHitShader = eClosestHit;
00242 shader_group_create_infos[3].anyHitShader = VK_SHADER_UNUSED_KHR;

```

```

00243     shader_group_create_infos[3].intersectionShader = VK_SHADER_UNUSED_KHR;
00244     shader_group_create_infos[3].pShaderGroupCaptureReplayHandle = nullptr;
00245
00246     shader_groups.push_back(shader_group_create_infos[3]);
00247
00248     VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00249     pipeline_layout_create_info.sType =
00250         VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00251     pipeline_layout_create_info.setLayoutCount =
00252         static_cast<uint32_t>(descriptorSetLayouts.size());
00253     pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00254     pipeline_layout_create_info.pushConstantRangeCount = 1;
00255     pipeline_layout_create_info.pPushConstantRanges = &pc_ranges;
00256
00257     VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00258                                             &pipeline_layout_create_info,
00259                                             nullptr, &pipeline_layout);
00260     ASSERT_VULKAN(result, "Failed to create raytracing pipeline layout!")
00261
00262     VkPipelineLibraryCreateInfoKHR pipeline_library_create_info{};
00263     pipeline_library_create_info.sType =
00264         VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR;
00265     pipeline_library_create_info.pNext = nullptr;
00266     pipeline_library_create_info.libraryCount = 0;
00267     pipeline_library_create_info.pLibraries = nullptr;
00268
00269     VkRayTracingPipelineCreateInfoKHR raytracing_pipeline_create_info{};
00270     raytracing_pipeline_create_info.sType =
00271         VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR;
00272     raytracing_pipeline_create_info.pNext = nullptr;
00273     raytracing_pipeline_create_info.flags = 0;
00274     raytracing_pipeline_create_info.stageCount =
00275         static_cast<uint32_t>(shader_stages.size());
00276     raytracing_pipeline_create_info.pStages = shader_stages.data();
00277     raytracing_pipeline_create_info.groupCount =
00278         static_cast<uint32_t>(shader_groups.size());
00279     raytracing_pipeline_create_info.pGroups = shader_groups.data();
00280     /*raytracing_pipeline_create_info.pLibraryInfo =
00281         &pipeline_library_create_info;
00282         raytracing_pipeline_create_info.pLibraryInterface = NULL;*/
00283 // TODO: HARDCODED FOR NOW;
00284     raytracing_pipeline_create_info.maxPipelineRayRecursionDepth = 2;
00285     raytracing_pipeline_create_info.layout = pipeline_layout;
00286
00287     result = pvkCreateRayTracingPipelinesKHR(
00288         device->getLogicalDevice(), VK_NULL_HANDLE, VK_NULL_HANDLE, 1,
00289         &raytracing_pipeline_create_info, nullptr, &graphicsPipeline);
00290
00291     ASSERT_VULKAN(result, "Failed to create raytracing pipeline!")
00292
00293     vkDestroyShaderModule(device->getLogicalDevice(), raygen_shader_module,
00294                           nullptr);
00295     vkDestroyShaderModule(device->getLogicalDevice(), raymiss_shader_module,
00296                           nullptr);
00297     vkDestroyShaderModule(device->getLogicalDevice(), raychit_shader_module,
00298                           nullptr);
00299     vkDestroyShaderModule(device->getLogicalDevice(), shadow_shader_module,
00300                           nullptr);
00301 }
00302
00303 void Raytracing::createSBT() {
00304     // load in functionality for raytracing shader group handles
00305     PFN_vkGetRayTracingShaderGroupHandlesKHR
00306         pvkGetRayTracingShaderGroupHandlesKHR =
00307             (PFN_vkGetRayTracingShaderGroupHandlesKHR)vkGetDeviceProcAddr(
00308                 device->getLogicalDevice(),
00309                 "vkGetRayTracingShaderGroupHandlesKHR");
00310
00311     raytracing_properties = VkPhysicalDeviceRayTracingPipelinePropertiesKHR{};
00312     raytracing_properties.sType =
00313         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR;
00314
00315     VkPhysicalDeviceProperties2 properties{};
00316     properties.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;
00317     properties.pNext = &raytracing_properties;
00318
00319     vkGetPhysicalDeviceProperties2(device->getPhysicalDevice(), &properties);
00320
00321     uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
00322     uint32_t handle_size_aligned =
00323         align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);
00324
00325     uint32_t group_count = static_cast<uint32_t>(shader_groups.size());
00326     uint32_t sbt_size = group_count * handle_size_aligned;
00327
00328     std::vector<uint8_t> handles(sbt_size);
00329

```

```

00330     VkResult result = pvkGetRayTracingShaderGroupHandlesKHR(
00331         device->getLogicalDevice(), graphicsPipeline, 0, group_count, sbt_size,
00332         handles.data());
00333     ASSERT_VULKAN(result, "Failed to get ray tracing shader group handles!")
00334
00335     const VkBufferUsageFlags bufferUsageFlags =
00336         VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR |
00337         VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT;
00338     const VkMemoryPropertyFlags memoryUsageFlags =
00339         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
00340         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT;
00341
00342     raygenShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
00343                                         memoryUsageFlags);
00344
00345     missShaderBindingTableBuffer.create(device, 2 * handle_size, bufferUsageFlags,
00346                                         memoryUsageFlags);
00347
00348     hitShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
00349                                         memoryUsageFlags);
00350
00351     void* mapped_raygen = nullptr;
00352     vkMapMemory(device->getLogicalDevice(),
00353                 raygenShaderBindingTableBuffer.getBufferMemory(), 0,
00354                 VK_WHOLE_SIZE, 0, &mapped_raygen);
00355
00356     void* mapped_miss = nullptr;
00357     vkMapMemory(device->getLogicalDevice(),
00358                 missShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
00359                 0, &mapped_miss);
00360
00361     void* mapped_rchit = nullptr;
00362     vkMapMemory(device->getLogicalDevice(),
00363                 hitShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
00364                 0, &mapped_rchit);
00365
00366     memcpy(mapped_raygen, handles.data(), handle_size);
00367     memcpy(mapped_miss, handles.data() + handle_size_aligned, handle_size * 2);
00368     memcpy(mapped_rchit, handles.data() + handle_size_aligned * 3, handle_size);
00369 }

```

## 6.183 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/← Src/renderer/VulkanRenderer.cpp File Reference

### 6.184 VulkanRenderer.cpp

[Go to the documentation of this file.](#)

```

00001 #include "VulkanRenderer.hpp"
00002
00003 #include <algorithm>
00004 #include <iostream>
00005 #include <vector>
00006
00007 #ifndef VMA_IMPLEMENTATION
00008 #define VMA_IMPLEMENTATION
00009 #endif // !VMA_IMPLEMENTATION
0010 #include <vk_mem_alloc.h>
0011
0012 #define STB_IMAGE_IMPLEMENTATION
0013 #include <stb_image.h>
0014
0015 #include <gsl/gsl>
0016
0017 #include "File.hpp"
0018 #include "Globals.hpp"
0019 #include "PushConstantPost.hpp"
0020 #include "ShaderHelper.hpp"
0021
0022 #include "VulkanRendererConfig.hpp"
0023
0024 VulkanRenderer::VulkanRenderer(Window* window, Scene* scene, GUI* gui,
0025                                     Camera* camera)
0026     :
0027
0028     window(window),
0029     scene(scene),
0030     gui(gui)

```

```
00031
00032 { updateUniforms(scene, camera, window);
00033
00034 try {
00035     instance = VulkanInstance();
00036
00037     VkDebugReportFlagsEXT debugReportFlags =
00038         VK_DEBUG_REPORT_ERROR_BIT_EXT | VK_DEBUG_REPORT_WARNING_BIT_EXT;
00039     if (ENABLE_VALIDATION_LAYERS)
00040         debug::setupDebugging(instance.getVulkanInstance(), debugReportFlags,
00041                               VK_NULL_HANDLE);
00042
00043     create_surface();
00044
00045     device = std::make_unique<VulkanDevice>(&instance, &surface);
00046
00047     allocator =
00048         Allocator(device->getLogicalDevice(), device->getPhysicalDevice(),
00049                   instance.getVulkanInstance());
00050
00051     create_command_pool();
00052
00053     vulkanSwapChain.initVulkanContext(device.get(), window, surface);
00054     create_uniform_buffers();
00055     create_command_buffers();
00056
00057     createSynchronization();
00058
00059     createSharedRenderDescriptorSetLayouts();
00060     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_rasterizer = {
00061         sharedRenderDescriptorSetLayout};
00062     rasterizer.init(device.get(), &vulkanSwapChain,
00063                      descriptor_set_layouts_rasterizer, graphics_command_pool);
00064     create_post_descriptor_layout();
00065     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
00066         post_descriptor_set_layout};
00067     postStage.init(device.get(), &vulkanSwapChain, descriptor_set_layouts_post);
00068     createDescriptorPoolSharedRenderStages();
00069     createSharedRenderDescriptorSet();
00070
00071     updatePostDescriptorSets();
00072
00073     createRaytracingDescriptorPool();
00074     createRaytracingDescriptorSetLayouts();
00075     std::vector<VkDescriptorSetLayout> layouts;
00076     layouts.push_back(sharedRenderDescriptorSetLayout);
00077     layouts.push_back(raytracingDescriptorSetLayout);
00078     raytracingStage.init(device.get(), layouts);
00079     pathTracing.init(device.get(), layouts);
00080
00081     scene->loadModel(device.get(), graphics_command_pool);
00082     updateTexturesInSharedRenderDescriptorSet();
00083
00084     asManager.createASForScene(device.get(), graphics_command_pool, scene);
00085     create_object_description_buffer();
00086     createRaytracingDescriptorSets();
00087     updateRaytracingDescriptorSets();
00088
00089     gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
00090                                    postStage.getRenderPass(),
00091                                    graphics_command_pool);
00092
00093 } catch (const std::runtime_error& e) {
00094     printf("ERROR: %s\n", e.what());
00095 }
00096 }
00097 }
00098
00099 void VulkanRenderer::updateUniforms(Scene* scene, Camera* camera,
00100                                         Window* window) {
00101     const GUISceneSharedVars guiSceneSharedVars = scene->getGuiSceneSharedVars();
00102
00103     globalUBO.view = camera->calculate_viewmatrix();
00104     globalUBO.projection =
00105         glm::perspective(glm::radians(camera->get_fov()),
00106                         (float)window->get_width() / (float)window->get_height(),
00107                         camera->get_near_plane(), camera->get_far_plane());
00108
00109     sceneUBO.view_dir = glm::vec4(camera->get_camera_direction(), 1.0f);
00110
00111     sceneUBO.light_dir =
00112         glm::vec4(guiSceneSharedVars.directional_light_direction[0],
00113                   guiSceneSharedVars.directional_light_direction[1],
00114                   guiSceneSharedVars.directional_light_direction[2], 1.0f);
00115
00116     sceneUBO.cam_pos =
00117         glm::vec4(camera->get_camera_position(), camera->get_fov());
```

```

0018 }
0019
0020 void VulkanRenderer::updateStateDueToUserInput(GUI* gui) {
0021     GUIRendererSharedVars& guiRendererSharedVars =
0022         gui->getGuiRendererSharedVars();
0023
0024     if (guiRendererSharedVars.shader_hot_reload_triggered) {
0025         shaderHotReload();
0026         guiRendererSharedVars.shader_hot_reload_triggered = false;
0027     }
0028 }
0029
0030 void VulkanRenderer::finishAllRenderCommands() {
0031     vkDeviceWaitIdle(device->getLogicalDevice());
0032 }
0033
0034 void VulkanRenderer::shaderHotReload() {
0035     // wait until no actions being run on device before destroying
0036     vkDeviceWaitIdle(device->getLogicalDevice());
0037
0038     std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
0039         sharedRenderDescriptorSetLayout};
0040     rasterizer.shaderHotReload(descriptor_set_layouts);
0041
0042     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
0043         post_descriptor_set_layout};
0044     postStage.shaderHotReload(descriptor_set_layouts_post);
0045
0046     std::vector<VkDescriptorSetLayout> layouts = {sharedRenderDescriptorSetLayout,
0047                                                 raytracingDescriptorSetLayout};
0048     raytracingStage.shaderHotReload(layouts);
0049     pathTracing.shaderHotReload(layouts);
0050 }
0051
0052 void VulkanRenderer::drawFrame() {
0053     // We need to skip one frame
0054     // Due to ImGui need to call ImGui::NewFrame() again
0055     // if we recreated swapchain
0056     if (checkChangedFramebufferSize()) return;
0057
0058     /*1. Get next available image to draw to and set something to signal when
0059      we're finished with the image (a semaphore) wait for given fence to signal
0060      (open) from last draw before continuing*/
0061     VkResult result = vkWaitForFences(device->getLogicalDevice(), 1,
0062                                         &in_flight_fences[current_frame], VK_TRUE,
0063                                         std::numeric_limits<uint64_t>::max());
0064     ASSERT_VULKAN(result, "Failed to wait for fences!")
0065     // -- GET NEXT IMAGE --
0066     uint32_t image_index;
0067     result = vkAcquireNextImageKHR(
0068         device->getLogicalDevice(), vulkanSwapChain.getSwapChain(),
0069         std::numeric_limits<uint64_t>::max(), image_available[current_frame],
0070         VK_NULL_HANDLE, &image_index);
0071
0072     if (result == VK_ERROR_OUT_OF_DATE_KHR) {
0073         // recreate_swap_chain();
0074         return;
0075     } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
0076         throw std::runtime_error("Failed to acquire next image!");
0077     }
0078
0079     //// check if previous frame is using this image (i.e. there is its fence to
0080     /// wait on)
0081     if (images_in_flight_fences[image_index] != VK_NULL_HANDLE) {
0082         vkWaitForFences(device->getLogicalDevice(), 1,
0083                         &images_in_flight_fences[image_index], VK_TRUE, UINT64_MAX);
0084     }
0085
0086     // mark the image as now being in use by this frame
0087     images_in_flight_fences[image_index] = in_flight_fences[current_frame];
0088
0089     VkCommandBufferBeginInfo buffer_begin_info{};
0090     buffer_begin_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
0091     buffer_begin_info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
0092     // start recording commands to command buffer
0093     result =
0094         vkBeginCommandBuffer(command_buffers[image_index], &buffer_begin_info);
0095     ASSERT_VULKAN(result, "Failed to start recording a command buffer!")
0096
0097     update_uniform_buffers(image_index);
0098
0099     GUIRendererSharedVars& guiRendererSharedVars =
0100         gui->getGuiRendererSharedVars();
0101     if (guiRendererSharedVars.raytracing)
0102         update_raytracing_descriptor_set(image_index);
0103
0104 }
```

```

00205     record_commands(image_index);
00206
00207     // stop recording to command buffer
00208     result = vkEndCommandBuffer(command_buffers[image_index]);
00209     ASSERT_VULKAN(result, "Failed to stop recording a command buffer!")
00210
00211     // 2. Submit command buffer to queue for execution, making sure it waits for
00212     // the image to be signalled as available before drawing and signals when it
00213     // has finished rendering
00214     // -- SUBMIT COMMAND BUFFER TO RENDER --
00215     VkSubmitInfo submit_info{};
00216     submit_info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
00217     submit_info.waitSemaphoreCount = 1; // number of semaphores to wait on
00218     submit_info.pWaitSemaphores =
00219         &image_available[current_frame]; // list of semaphores to wait on
00220
00221     VkPipelineStageFlags wait_stages = {
00222
00223         VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT /*|
00224             VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT |
00225             VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR*/
00226     };
00227
00228     submit_info.pWaitDstStageMask =
00229         &wait_stages; // stages to check semaphores at
00230
00231     submit_info.commandBufferCount = 1; // number of command buffers to submit
00232     submit_info.pCommandBuffers =
00233         &command_buffers[image_index]; // command buffer to submit
00234     submit_info.signalSemaphoreCount = 1; // number of semaphores to signal
00235     submit_info.pSignalSemaphores =
00236         &render_finished[current_frame]; // semaphores to signal when command
00237         // buffer finishes
00238
00239     result = vkResetFences(device->getLogicalDevice(), 1,
00240         &in_flight_fences[current_frame]);
00241     ASSERT_VULKAN(result, "Failed to reset fences!")
00242
00243     // submit command buffer to queue
00244     result = vkQueueSubmit(device->getGraphicsQueue(), 1, &submit_info,
00245         in_flight_fences[current_frame]);
00246     ASSERT_VULKAN(result, "Failed to submit command buffer to queue!")
00247
00248     // 3. Present image to screen when it has signalled finished rendering
00249     // -- PRESENT RENDERED IMAGE TO SCREEN --
00250     VkPresentInfoKHR present_info{};
00251     present_info.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
00252     present_info.waitSemaphoreCount = 1; // number of semaphores to wait on
00253     present_info.pWaitSemaphores =
00254         &render_finished[current_frame]; // semaphores to wait on
00255     present_info.swapChainCount = 1; // number of swapchains to present to
00256     const VkSwapchainKHR swapchain = vulkanSwapChain.getSwapChain();
00257     present_info.pSwapchains = &swapchain; // swapchains to present images to
00258     present_info.pImageIndices =
00259         &image_index; // index of images in swapchain to present
00260
00261     result = vkQueuePresentKHR(device->getPresentationQueue(), &present_info);
00262
00263     if (result == VK_ERROR_OUT_OF_DATE_KHR) {
00264         // recreate_swap_chain();
00265         return;
00266     }
00267
00268     } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
00269         throw std::runtime_error("Failed to acquire next image!");
00270     }
00271
00272     if (result != VK_SUCCESS) {
00273         throw std::runtime_error("Failed to submit to present queue!");
00274     }
00275
00276     current_frame = (current_frame + 1) % MAX_FRAME_DRAWS;
00277 }
00278
00279 void VulkanRenderer::create_surface() {
00280     // create surface (creates a surface create info struct, runs the create
00281     // surface function, returns result)
00282     ASSERT_VULKAN(
00283         glfwCreateWindowSurface(instance.getVulkanInstance(),
00284             window->get_window(), nullptr, &surface),
00285         "Failed to create a surface!");
00286 }
00287
00288 void VulkanRenderer::create_post_descriptor_layout() {
00289     // UNIFORM VALUES DESCRIPTOR SET LAYOUT
00290     // globalUBO Binding info
00291     VkDescriptorSetLayoutBinding post_sampler_layout_binding{};


```

```

00292     post_sampler_layout_binding.binding =
00293         0; // binding point in shader (designated by binding number in shader)
00294     post_sampler_layout_binding.descriptorType =
00295         VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER; // type of descriptor
00296         // (uniform, dynamic uniform,
00297         // image sampler, etc)
00298     post_sampler_layout_binding.descriptorCount =
00299         1; // number of descriptors for binding
00300     post_sampler_layout_binding.stageFlags =
00301         VK_SHADER_STAGE_FRAGMENT_BIT; // we need to say at which shader we bind
00302         // this uniform to
00303     post_sampler_layout_binding.pImmutableSamplers =
00304         nullptr; // for texture: can make sampler data unchangeable (immutable)
00305         // by specifying in layout
00306
00307     std::vector<VkDescriptorSetLayoutBinding> layout_bindings = {
00308         post_sampler_layout_binding};
00309
00310     // create descriptor set layout with given bindings
00311     VkDescriptorSetLayoutCreateInfo layout_create_info{};
00312     layout_create_info.sType =
00313         VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00314     layout_create_info.bindingCount = static_cast<uint32_t>(
00315         layout_bindings.size()); // only have 1 for the globalUBO
00316     layout_create_info.pBindings =
00317         layout_bindings.data(); // array of binding infos
00318
00319     // create descriptor set layout
00320     VkResult result = vkCreateDescriptorSetLayout(device->getLogicalDevice(),
00321             &layout_create_info, nullptr,
00322             &post_descriptor_set_layout);
00323     ASSERT_VULKAN(result, "Failed to create descriptor set layout!")
00324
00325     VkDescriptorPoolSize post_pool_size{};
00326     post_pool_size.type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
00327     post_pool_size.descriptorCount = static_cast<uint32_t>(1);
00328
00329     // list of pool sizes
00330     std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {post_pool_size};
00331
00332     VkDescriptorPoolCreateInfo pool_create_info{};
00333     pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00334     pool_create_info.maxSets =
00335         vulkanSwapChain
00336             .getNumberSwapChainImages(); // maximum number of descriptor sets
00337             // that can be created from pool
00338     pool_create_info.poolSizeCount = static_cast<uint32_t>(
00339         descriptor_pool_sizes.size()); // amount of pool sizes being passed
00340     pool_create_info.pPoolSizes =
00341         descriptor_pool_sizes.data(); // pool sizes to create pool with
00342
00343     // create descriptor pool
00344     result = vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
00345             nullptr, &post_descriptor_pool);
00346     ASSERT_VULKAN(result, "Failed to create a descriptor pool!")
00347
00348     // resize descriptor set list so one for every buffer
00349     post_descriptor_set.resize(vulkanSwapChain.getNumberSwapChainImages());
00350
00351     std::vector<VkDescriptorSetLayout> set_layouts(
00352         vulkanSwapChain.getNumberSwapChainImages(), post_descriptor_set_layout);
00353
00354     // descriptor set allocation info
00355     VkDescriptorSetAllocateInfo set_alloc_info{};
00356     set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00357     set_alloc_info.descriptorPool =
00358         post_descriptor_pool; // pool to allocate descriptor set from
00359     set_alloc_info.descriptorSetCount =
00360         vulkanSwapChain.getNumberSwapChainImages(); // number of sets to allocate
00361     set_alloc_info.pSetLayouts =
00362         set_layouts.data(); // layouts to use to allocate sets (1:1 relationship)
00363
00364     // allocate descriptor sets (multiple)
00365     result = vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
00366             post_descriptor_set.data());
00367     ASSERT_VULKAN(result, "Failed to create descriptor sets!")
00368 }
00369
00370 void VulkanRenderer::updatePostDescriptorSets() {
00371     // update all of descriptor set buffer bindings
00372     for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00373         // texture image info
00374         VkDescriptorImageInfo image_info{};
00375         image_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00376         Texture& renderResult = rasterizer.getOffscreenTexture(i);
00377         image_info.imageView = renderResult.getImageView();
00378         image_info.sampler = postStage.getOffscreenSampler();

```

```

00379 // descriptor write info
00380 VkWriteDescriptorSet descriptor_write{};
00381 descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00382 descriptor_write.dstSet = post_descriptor_set[i];
00383 descriptor_write.dstBinding = 0;
00384 descriptor_write.dstArrayElement = 0;
00385 descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
00386 descriptor_write.descriptorCount = 1;
00387 descriptor_write.pImageInfo = &image_info;
00388
00389 // update new descriptor set
00390 vkUpdateDescriptorSets(device->getLogicalDevice(), 1, &descriptor_write, 0,
00391                         nullptr);
00392 }
00393 }
00394 }
00395
00396 void VulkanRenderer::createRaytracingDescriptorPool() {
00397     std::array<VkDescriptorPoolSize, 2> descriptor_pool_sizes{};
00398
00399     descriptor_pool_sizes[0].type = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00400     descriptor_pool_sizes[0].descriptorCount = 1;
00401
00402     descriptor_pool_sizes[1].type = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00403     descriptor_pool_sizes[1].descriptorCount = 1;
00404
00405     VkDescriptorPoolCreateInfo descriptor_pool_create_info{};
00406     descriptor_pool_create_info.sType =
00407         VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00408     descriptor_pool_create_info.poolSizeCount =
00409         static_cast<uint32_t>(descriptor_pool_sizes.size());
00410     descriptor_pool_create_info.pPoolSizes = descriptor_pool_sizes.data();
00411     descriptor_pool_create_info.maxSets =
00412         vulkanSwapChain.getNumberOfSwapChainImages();
00413
00414     VkResult result = vkCreateDescriptorPool(device->getLogicalDevice(),
00415                                              &descriptor_pool_create_info,
00416                                              nullptr, &raytracingDescriptorPool);
00417     ASSERT_VULKAN(result, "Failed to create command pool!")
00418 }
00419
00420 void VulkanRenderer::cleanUpSync() {
00421     for (int i = 0; i < MAX_FRAME_DRAWS; i++) {
00422         vkDestroySemaphore(device->getLogicalDevice(), render_finished[i], nullptr);
00423         vkDestroySemaphore(device->getLogicalDevice(), image_available[i], nullptr);
00424         vkDestroyFence(device->getLogicalDevice(), in_flight_fences[i], nullptr);
00425     }
00426 }
00427
00428 void VulkanRenderer::create_object_description_buffer() {
00429     std::vector<ObjectDescription> objectDescriptions =
00430         scene->getObjectDescriptions();
00431
00432     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00433         device.get(), graphics_command_pool, objectDescriptionBuffer,
00434         VK_BUFFER_USAGE_TRANSFER_DST_BIT |
00435             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00436             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00437             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00438             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00439         objectDescriptions);
00440
00441 // update the object description set
00442 // update all of descriptor set buffer bindings
00443 for (size_t i = 0; i < vulkanSwapChain.getNumberOfSwapChainImages(); i++) {
00444     VkDescriptorBufferInfo object_descriptions_buffer_info{};
00445     // image_info.sampler = VK_DESCRIPTOR_TYPE_SAMPLER;
00446     object_descriptions_buffer_info.buffer =
00447         objectDescriptionBuffer.getBuffer();
00448     object_descriptions_buffer_info.offset = 0;
00449     object_descriptions_buffer_info.range = VK_WHOLE_SIZE;
00450
00451     VkWriteDescriptorSet descriptor_object_descriptions_writer{};
00452     descriptor_object_descriptions_writer.sType =
00453         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00454     descriptor_object_descriptions_writer.pNext = nullptr;
00455     descriptor_object_descriptions_writer.dstSet = sharedRenderDescriptorSet[i];
00456     descriptor_object_descriptions_writer.dstBinding =
00457         OBJECT_DESCRIPTION_BINDING;
00458     descriptor_object_descriptions_writer.dstArrayElement = 0;
00459     descriptor_object_descriptions_writer.descriptorCount = 1;
00460     descriptor_object_descriptions_writer.descriptorType =
00461         VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00462     descriptor_object_descriptions_writer.pImageInfo = nullptr;
00463     descriptor_object_descriptions_writer.pBufferInfo =
00464         &object_descriptions_buffer_info;
00465     descriptor_object_descriptions_writer.pTexelBufferView =

```

```

00466     nullptr; // information about buffer data to bind
00467
00468     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00469         descriptor_object_descriptions_writer};
00470
00471     // update the descriptor sets with new buffer/binding info
00472     vkUpdateDescriptorSets(device->getLogicalDevice(),
00473                             static_cast<uint32_t>(write_descriptor_sets.size()),
00474                             write_descriptor_sets.data(), 0, nullptr);
00475 }
00476 }
00477
00478 void VulkanRenderer::createRaytracingDescriptorSetLayouts() {
00479 {
00480     std::array<VkDescriptorSetLayoutBinding, 2>
00481         descriptor_set_layout_bindings{};
00482
00483     // here comes the top level acceleration structure
00484     descriptor_set_layout_bindings[0].binding = TIAS_BINDING;
00485     descriptor_set_layout_bindings[0].descriptorCount = 1;
00486     descriptor_set_layout_bindings[0].descriptorType =
00487         VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00488     descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;
00489     // load them into the raygeneration and closest hit shader
00490     descriptor_set_layout_bindings[0].stageFlags =
00491         VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00492         VK_SHADER_STAGE_COMPUTE_BIT;
00493     // here comes to previous rendered image
00494     descriptor_set_layout_bindings[1].binding = OUT_IMAGE_BINDING;
00495     descriptor_set_layout_bindings[1].descriptorCount = 1;
00496     descriptor_set_layout_bindings[1].descriptorType =
00497         VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00498     descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;
00499     // load them into the raygeneration and closest hit shader
00500     descriptor_set_layout_bindings[1].stageFlags =
00501         VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00502         VK_SHADER_STAGE_COMPUTE_BIT;
00503
00504     VkDescriptorSetLayoutCreateInfo descriptor_set_layout_create_info{};
00505     descriptor_set_layout_create_info.sType =
00506         VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00507     descriptor_set_layout_create_info.bindingCount =
00508         static_cast<uint32_t>(descriptor_set_layout_bindings.size());
00509     descriptor_set_layout_create_info.pBindings =
00510         descriptor_set_layout_bindings.data();
00511
00512     VkResult result = vkCreateDescriptorSetLayout(
00513         device->getLogicalDevice(), &descriptor_set_layout_create_info, nullptr,
00514         &raytracingDescriptorSetLayout);
00515     ASSERT_VULKAN(result, "Failed to create raytracing descriptor set layout!")
00516 }
00517 }
00518
00519 void VulkanRenderer::createRaytracingDescriptorSets() {
00520     // resize descriptor set list so one for every buffer
00521     raytracingDescriptorSet.resize(vulkanSwapChain.getNumberSwapChainImages());
00522
00523     std::vector<VkDescriptorSetLayout> set_layouts(
00524         vulkanSwapChain.getNumberSwapChainImages(),
00525         raytracingDescriptorSetLayout);
00526
00527     VkDescriptorSetAllocateInfo descriptor_set_allocate_info{};
00528     descriptor_set_allocate_info.sType =
00529         VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00530 ;
00531     descriptor_set_allocate_info.descriptorPool = raytracingDescriptorPool;
00532     descriptor_set_allocate_info.descriptorSetCount =
00533         vulkanSwapChain.getNumberSwapChainImages();
00534     descriptor_set_allocate_info.pSetLayouts = set_layouts.data();
00535
00536     VkResult result = vkAllocateDescriptorSets(device->getLogicalDevice(),
00537                                             &descriptor_set_allocate_info,
00538                                             raytracingDescriptorSet.data());
00539     ASSERT_VULKAN(result, "Failed to allocate raytracing descriptor set!")
00540 }
00541
00542 void VulkanRenderer::updateRaytracingDescriptorSets() {
00543     for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00544         VkWriteDescriptorSetAccelerationStructureKHR
00545             descriptor_set_acceleration_structure{};
00546         descriptor_set_acceleration_structure.sType =
00547             VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
00548         descriptor_set_acceleration_structure.pNext = nullptr;
00549         descriptor_set_acceleration_structure.accelerationStructureCount = 1;
00550         VkAccelerationStructureKHR& vulkanTIAS = asManager.getTIAS();
00551         descriptor_set_acceleration_structure.pAccelerationStructures = &vulkanTIAS;
00552

```

```
00553     VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
00554     write_descriptor_set_acceleration_structure.sType =
00555         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00556     write_descriptor_set_acceleration_structure.pNext =
00557         &descriptor_set_acceleration_structure;
00558     write_descriptor_set_acceleration_structure.dstSet =
00559         raytracingDescriptorSet[i];
00560     write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
00561     write_descriptor_set_acceleration_structure.dstArrayElement = 0;
00562     write_descriptor_set_acceleration_structure.descriptorCount = 1;
00563     write_descriptor_set_acceleration_structure.descriptorType =
00564         VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00565     write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
00566     write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
00567     write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;
00568
00569     VkDescriptorImageInfo image_info{};
00570     Texture& renderResult = rasterizer.getOffscreenTexture(i);
00571     image_info.imageView = renderResult.getImageView();
00572     image_info.imageLayout = VK_IMAGE_LAYOUT_GENERAL;
00573
00574     VkWriteDescriptorSet descriptor_image_writer{};
00575     descriptor_image_writer.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00576     descriptor_image_writer.pNext = nullptr;
00577     descriptor_image_writer.dstSet = raytracingDescriptorSet[i];
00578     descriptor_image_writer.dstBinding = OUT_IMAGE_BINDING;
00579     descriptor_image_writer.dstArrayElement = 0;
00580     descriptor_image_writer.descriptorCount = 1;
00581     descriptor_image_writer.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00582     descriptor_image_writer.pImageInfo = &image_info;
00583     descriptor_image_writer.pBufferInfo = nullptr;
00584     descriptor_image_writer.pTexelBufferView = nullptr;
00585
00586     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00587         write_descriptor_set_acceleration_structure, descriptor_image_writer};
00588
00589     // update the descriptor sets with new buffer/binding info
00590     vkUpdateDescriptorSets(device->getLogicalDevice(),
00591         static_cast<uint32_t>(write_descriptor_sets.size()),
00592         write_descriptor_sets.data(), 0, nullptr);
00593 }
00594 }
00595
00596 void VulkanRenderer::createSharedRenderDescriptorsetLayouts() {
00597     std::array<VkDescriptorSetLayoutBinding, 5> descriptor_set_layout_bindings{};
00598     // UNIFORM VALUES DESCRIPTOR SET LAYOUT
00599     // globalUBO Binding info
00600     descriptor_set_layout_bindings[0].binding = globalUBO_BINDING;
00601     descriptor_set_layout_bindings[0].descriptorType =
00602         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00603     descriptor_set_layout_bindings[0].descriptorCount = 1;
00604     descriptor_set_layout_bindings[0].stageFlags =
00605         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00606         VK_SHADER_STAGE_COMPUTE_BIT;
00607     descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;
00608
00609     // our model matrix which updates every frame for each object
00610     descriptor_set_layout_bindings[1].binding = sceneUBO_BINDING;
00611     descriptor_set_layout_bindings[1].descriptorType =
00612         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00613     descriptor_set_layout_bindings[1].descriptorCount = 1;
00614     descriptor_set_layout_bindings[1].stageFlags =
00615         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
00616         VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00617         VK_SHADER_STAGE_COMPUTE_BIT;
00618     descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;
00619
00620     descriptor_set_layout_bindings[2].binding = OBJECT_DESCRIPTION_BINDING;
00621     descriptor_set_layout_bindings[2].descriptorCount = 1;
00622     descriptor_set_layout_bindings[2].descriptorType =
00623         VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00624     descriptor_set_layout_bindings[2].pImmutableSamplers = nullptr;
00625     // load them into the raygeneration and chlosest hit shader
00626     descriptor_set_layout_bindings[2].stageFlags =
00627         VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
00628         VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR | VK_SHADER_STAGE_COMPUTE_BIT;
00629
00630     // CREATE TEXTURE SAMPLER DESCRIPTOR SET LAYOUT
00631     // texture binding info
00632     descriptor_set_layout_bindings[3].binding = SAMPLER_BINDING;
00633     descriptor_set_layout_bindings[3].descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
00634     descriptor_set_layout_bindings[3].descriptorCount = MAX_TEXTURE_COUNT;
00635     descriptor_set_layout_bindings[3].stageFlags =
00636         VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00637         VK_SHADER_STAGE_COMPUTE_BIT;
00638     descriptor_set_layout_bindings[3].pImmutableSamplers = nullptr;
00639 }
```

```

00640     descriptor_set_layout_bindings[4].binding = TEXTURES_BINDING;
00641     descriptor_set_layout_bindings[4].descriptorType =
00642         VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00643     descriptor_set_layout_bindings[4].descriptorCount = MAX_TEXTURE_COUNT;
00644     descriptor_set_layout_bindings[4].stageFlags =
00645         VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00646         VK_SHADER_STAGE_COMPUTE_BIT;
00647     descriptor_set_layout_bindings[4].pImmutableSamplers = nullptr;
00648
00649 // create descriptor set layout with given bindings
00650 VkDescriptorSetLayoutCreateInfo layout_create_info{};
00651 layout_create_info.sType =
00652     VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00653 layout_create_info.bindingCount =
00654     static_cast<uint32_t>(descriptor_set_layout_bindings.size());
00655 layout_create_info.pBindings = descriptor_set_layout_bindings.data();
00656
00657 // create descriptor set layout
00658 VkResult result = vkCreateDescriptorSetLayout(
00659     device->getLogicalDevice(), &layout_create_info, nullptr,
00660     &sharedRenderDescriptorSetLayout);
00661 ASSERT_VULKAN(result, "Failed to create descriptor set layout!")
00662 }
00663
00664 void VulkanRenderer::create_command_pool() {
00665     // get indices of queue families from device
00666     QueueFamilyIndices queue_family_indices = device->getQueueFamilies();
00667
00668 {
00669     VkCommandPoolCreateInfo pool_info{};
00670     pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
00671     pool_info.flags =
00672         VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT; // we are ready now to
00673                                         // re-record our
00674                                         // command buffers
00675     pool_info.queueFamilyIndex =
00676         queue_family_indices
00677             .graphics_family; // queue family type that buffers from this
00678             // command pool will use
00679
00680     // create a graphics queue family command pool
00681     VkResult result =
00682         vkCreateCommandPool(device->getLogicalDevice(), &pool_info, nullptr,
00683             &graphics_command_pool);
00684     ASSERT_VULKAN(result, "Failed to create command pool!")
00685 }
00686
00687 {
00688     VkCommandPoolCreateInfo pool_info{};
00689     pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
00690     pool_info.flags =
00691         VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT; // we are ready now to
00692                                         // re-record our
00693                                         // command buffers
00694     pool_info.queueFamilyIndex =
00695         queue_family_indices.compute_family; // queue family type that buffers
00696             // from this command pool will use
00697
00698     // create a graphics queue family command pool
00699     VkResult result = vkCreateCommandPool(
00700         device->getLogicalDevice(), &pool_info, nullptr, &compute_command_pool);
00701     ASSERT_VULKAN(result, "Failed to create command pool!")
00702 }
00703 }
00704
00705 void VulkanRenderer::cleanUpCommandPools() {
00706     vkDestroyCommandPool(device->getLogicalDevice(), graphics_command_pool,
00707         nullptr);
00708     vkDestroyCommandPool(device->getLogicalDevice(), compute_command_pool,
00709         nullptr);
00710 }
00711
00712 void VulkanRenderer::create_command_buffers() {
00713     // resize command buffer count to have one for each framebuffer
00714     command_buffers.resize(vulkanSwapChain.getNumberOfSwapChainImages());
00715
00716     VkCommandBufferAllocateInfo command_buffer_alloc_info{};
00717     command_buffer_alloc_info.sType =
00718         VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
00719     command_buffer_alloc_info.commandPool = graphics_command_pool;
00720     command_buffer_alloc_info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
00721
00722     command_buffer_alloc_info.commandBufferCount =
00723         static_cast<uint32_t>(command_buffers.size());
00724
00725     VkResult result = vkAllocateCommandBuffers(device->getLogicalDevice(),
00726         &command_buffer_alloc_info,

```

```

00727                                     command_buffers.data());
00728     ASSERT_VULKAN(result, "Failed to allocate command buffers!")
00729 }
00730
00731 void VulkanRenderer::createSynchronization() {
00732     image_available.resize(vulkanSwapChain.getNumberSwapChainImages(),
00733                             VK_NULL_HANDLE);
00734     render_finished.resize(vulkanSwapChain.getNumberSwapChainImages(),
00735                             VK_NULL_HANDLE);
00736     in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
00737                             VK_NULL_HANDLE);
00738     images_in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
00739                                     VK_NULL_HANDLE);
00740
00741     // semaphore creation information
00742     VkSemaphoreCreateInfo semaphore_create_info{};
00743     semaphore_create_info.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
00744
00745     // fence creation information
00746     VkFenceCreateInfo fence_create_info{};
00747     fence_create_info.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
00748     fence_create_info.flags = VK_FENCE_CREATE_SIGNALED_BIT;
00749
00750     for (int i = 0; i < MAX_FRAME_DRAWS; i++) {
00751         if ((vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
00752                               nullptr, &image_available[i]) != VK_SUCCESS) ||
00753             (vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
00754                               nullptr, &render_finished[i]) != VK_SUCCESS) ||
00755             (vkCreateFence(device->getLogicalDevice(), &fence_create_info, nullptr,
00756                           &in_flight_fences[i]) != VK_SUCCESS)) {
00757             throw std::runtime_error("Failed to create a semaphore and/or fence!");
00758         }
00759     }
00760 }
00761
00762 void VulkanRenderer::create_uniform_buffers() {
00763     // one uniform buffer for each image (and by extension, command buffer)
00764     globalUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());
00765     sceneUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());
00766
00767     // temporary buffer to "stage" vertex data before transferring to GPU
00768     // VulkanBuffer stagingBuffer;
00769     std::vector<GlobalUBO> globalUBOdata;
00770     globalUBOdata.push_back(globalUBO);
00771
00772     std::vector<SceneUBO> sceneUBOdata;
00773     sceneUBOdata.push_back(sceneUBO);
00774
00775     // create uniform buffers
00776     for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00777         vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00778             device.get(), graphics_command_pool, globalUBOBuffer[i],
00779             VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00780             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, globalUBOdata);
00781
00782         vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00783             device.get(), graphics_command_pool, sceneUBOBuffer[i],
00784             VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00785             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, sceneUBOdata);
00786     }
00787 }
00788
00789 void VulkanRenderer::createDescriptorPoolSharedRenderStages() {
00790     // CREATE UNIFORM DESCRIPTOR POOL
00791     // type of descriptors + how many descriptors, not descriptor sets (combined
00792     // makes the pool size) ViewProjection Pool
00793     VkDescriptorPoolSize vp_pool_size{};
00794     vp_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00795     vp_pool_size.descriptorCount = static_cast<uint32_t>(globalUBOBuffer.size());
00796
00797     // DIRECTION POOL
00798     VkDescriptorPoolSize directions_pool_size{};
00799     directions_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00800     directions_pool_size.descriptorCount =
00801         static_cast<uint32_t>(sceneUBOBuffer.size());
00802
00803     VkDescriptorPoolSize object_descriptions_pool_size{};
00804     object_descriptions_pool_size.type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00805     object_descriptions_pool_size.descriptorCount =
00806         static_cast<uint32_t>(sizeof(ObjectDescription) * MAX_OBJECTS);
00807
00808     // TEXTURE SAMPLER POOL
00809     VkDescriptorPoolSize sampler_pool_size{};
00810     sampler_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLER;
00811     sampler_pool_size.descriptorCount = MAX_TEXTURE_COUNT;
00812
00813     VkDescriptorPoolSize sampled_image_pool_size{};

```

```

00814     sampled_image_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00815     sampled_image_pool_size.descriptorCount = MAX_TEXTURE_COUNT;
00816
00817     // list of pool sizes
00818     std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {
00819         vp_pool_size, directions_pool_size, object_descriptions_pool_size,
00820         sampler_pool_size, sampled_image_pool_size};
00821
00822     VkDescriptorPoolCreateInfo pool_create_info{};
00823     pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00824     pool_create_info.maxSets =
00825         vulkanSwapChain
00826             .getNumberSwapChainImages(); // maximum number of descriptor sets
00827                                         // that can be created from pool
00828     pool_create_info.poolSizeCount = static_cast<uint32_t>(
00829         descriptor_pool_sizes.size()); // amount of pool sizes being passed
00830     pool_create_info.pPoolSizes =
00831         descriptor_pool_sizes.data(); // pool sizes to create pool with
00832
00833     // create descriptor pool
00834     VkResult result =
00835         vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
00836                               nullptr, &descriptorPoolSharedRenderStages);
00837     ASSERT_VULKAN(result, "Failed to create a descriptor pool!")
00838 }
00839
00840 void VulkanRenderer::createSharedRenderDescriptorSet() {
00841     // resize descriptor set list so one for every buffer
00842     sharedRenderDescriptorSet.resize(vulkanSwapChain.getNumberOfSwapChainImages());
00843
00844     std::vector<VkDescriptorSetLayout> set_layouts(
00845         vulkanSwapChain.getNumberOfSwapChainImages(),
00846         sharedRenderDescriptorSetLayout);
00847
00848     // descriptor set allocation info
00849     VkDescriptorSetAllocateInfo set_alloc_info{};
00850     set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00851     set_alloc_info.descriptorPool =
00852         descriptorPoolSharedRenderStages; // pool to allocate descriptor set from
00853     set_alloc_info.descriptorSetCount =
00854         vulkanSwapChain.getNumberOfSwapChainImages(); // number of sets to allocate
00855     set_alloc_info.pSetLayouts =
00856         set_layouts.data(); // layouts to use to allocate sets (1:1 relationship)
00857
00858     // allocate descriptor sets (multiple)
00859     VkResult result =
00860         vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
00861                               sharedRenderDescriptorSet.data());
00862     ASSERT_VULKAN(result, "Failed to create descriptor sets!")
00863
00864     // update all of descriptor set buffer bindings
00865     for (size_t i = 0; i < vulkanSwapChain.getNumberOfSwapChainImages(); i++) {
00866         // VIEW PROJECTION DESCRIPTOR
00867         // buffer info and data offset info
00868         VkDescriptorBufferInfo globalUBO_buffer_info{};
00869         globalUBO_buffer_info.buffer =
00870             globalUBOBuffer[i].getBuffer(); // buffer to get data from
00871         globalUBO_buffer_info.offset = 0; // position of start of data
00872         globalUBO_buffer_info.range = sizeof(globalUBO); // size of data
00873
00874         // data about connection between binding and buffer
00875         VkWriteDescriptorSet globalUBO_set_write{};
00876         globalUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00877         globalUBO_set_write.dstSet =
00878             sharedRenderDescriptorSet[i]; // descriptor set to update
00879         globalUBO_set_write.dstBinding =
00880             0; // binding to update (matches with binding on layout/shader)
00881         globalUBO_set_write.dstArrayElement = 0; // index in array to update
00882         globalUBO_set_write.descriptorType =
00883             VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER; // type of descriptor
00884         globalUBO_set_write.descriptorCount = 1; // amount to update
00885         globalUBO_set_write.pBufferInfo =
00886             &globalUBO_buffer_info; // information about buffer data to bind
00887
00888         // VIEW PROJECTION DESCRIPTOR
00889         // buffer info and data offset info
00890         VkDescriptorBufferInfo sceneUBO_buffer_info{};
00891         sceneUBO_buffer_info.buffer =
00892             sceneUBOBuffer[i].getBuffer(); // buffer to get data from
00893         sceneUBO_buffer_info.offset = 0; // position of start of data
00894         sceneUBO_buffer_info.range = sizeof(sceneUBO); // size of data
00895
00896         // data about connection between binding and buffer
00897         VkWriteDescriptorSet sceneUBO_set_write{};
00898         sceneUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00899         sceneUBO_set_write.dstSet =
00900             sharedRenderDescriptorSet[i]; // descriptor set to update

```

```

00901     sceneUBO_set_write.dstBinding =
00902         1; // binding to update (matches with binding on layout/shader)
00903     sceneUBO_set_write.dstArrayElement = 0; // index in array to update
00904     sceneUBO_set_write.descriptorType =
00905         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER; // type of descriptor
00906     sceneUBO_set_write.descriptorCount = 1; // amount to update
00907     sceneUBO_set_write.pBufferInfo =
00908         &sceneUBO_buffer_info; // information about buffer data to bind
00909
00910     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00911         globalUBO_set_write, sceneUBO_set_write};
00912
00913     // update the descriptor sets with new buffer/binding info
00914     vkUpdateDescriptorSets(device->getLogicalDevice(),
00915         static_cast<uint32_t>(write_descriptor_sets.size()),
00916         write_descriptor_sets.data(), 0, nullptr);
00917 }
00918 }
00919
00920 void VulkanRenderer::updateTexturesInSharedRenderDescriptorSet() {
00921     std::vector<Texture>& modelTextures = scene->getTextures(0);
00922     std::vector<VkDescriptorImageInfo> image_info_textures;
00923     image_info_textures.resize(scene->getTextureCount(0));
00924     for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
00925         image_info_textures[i].imageLayout =
00926             VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00927         image_info_textures[i].imageView = modelTextures[i].getImageView();
00928         image_info_textures[i].sampler = nullptr;
00929     }
00930
00931     std::vector<VkSampler>& modelTextureSampler = scene->getTextureSampler(0);
00932     std::vector<VkDescriptorImageInfo> image_info_texture_sampler;
00933     image_info_texture_sampler.resize(scene->getTextureCount(0));
00934     for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
00935         image_info_texture_sampler[i].imageLayout =
00936             VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00937         image_info_texture_sampler[i].imageView = nullptr;
00938         image_info_texture_sampler[i].sampler = modelTextureSampler[i];
00939     }
00940
00941     for (uint32_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00942         // descriptor write info
00943         VkWriteDescriptorSet descriptor_write{};
00944         descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00945         descriptor_write.dstSet = sharedRenderDescriptorSet[i];
00946         descriptor_write.dstBinding = TEXTURES_BINDING;
00947         descriptor_write.dstArrayElement = 0;
00948         descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00949         descriptor_write.descriptorCount =
00950             static_cast<uint32_t>(image_info_textures.size());
00951         descriptor_write.pImageInfo = image_info_textures.data();
00952
00953         /*VkDescriptorImageInfo sampler_info;
00954             sampler_info.imageView = nullptr;
00955             sampler_info.sampler = texture_sampler;*/
00956
00957         // descriptor write info
00958         VkWriteDescriptorSet descriptor_write_sampler{};
00959         descriptor_write_sampler.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00960         descriptor_write_sampler.dstSet = sharedRenderDescriptorSet[i];
00961         descriptor_write_sampler.dstBinding = SAMPLER_BINDING;
00962         descriptor_write_sampler.dstArrayElement = 0;
00963         descriptor_write_sampler.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
00964         descriptor_write_sampler.descriptorCount =
00965             static_cast<uint32_t>(image_info_texture_sampler.size());
00966         descriptor_write_sampler.pImageInfo = image_info_texture_sampler.data();
00967
00968         std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00969             descriptor_write, descriptor_write_sampler};
00970
00971         // update new descriptor set
00972         vkUpdateDescriptorSets(device->getLogicalDevice(),
00973             static_cast<uint32_t>(write_descriptor_sets.size()),
00974             write_descriptor_sets.data(), 0, nullptr);
00975     }
00976 }
00977
00978 void VulkanRenderer::cleanUpUBOs() {
00979     for (VulkanBuffer vulkanBuffer : globalUBOBuffer) {
00980         vulkanBuffer.cleanUp();
00981     }
00982
00983     for (VulkanBuffer vulkanBuffer : sceneUBOBuffer) {
00984         vulkanBuffer.cleanUp();
00985     }
00986 }
00987

```

```

00988 void VulkanRenderer::update_uniform_buffers(uint32_t image_index) {
00989     auto usage_stage_flags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
00990                             VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR |
00991                             VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
00992
00993     VkBufferMemoryBarrier before_barrier_uvp{};
00994     before_barrier_uvp.pNext = nullptr;
00995     before_barrier_uvp.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
00996     before_barrier_uvp.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
00997     before_barrier_uvp.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00998     before_barrier_uvp.buffer = globalUBOBuffer[image_index].getBuffer();
00999     before_barrier_uvp.offset = 0;
01000     before_barrier_uvp.size = sizeof(globalUBO);
01001     before_barrier_uvp.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01002     before_barrier_uvp.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01003
01004     VkBufferMemoryBarrier before_barrier_directions{};
01005     before_barrier_directions.pNext = nullptr;
01006     before_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01007     before_barrier_directions.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
01008     before_barrier_directions.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01009     before_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
01010     before_barrier_directions.offset = 0;
01011     before_barrier_directions.size = sizeof(sceneUBO);
01012     before_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01013     before_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01014
01015     vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
01016                           VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
01017                           &before_barrier_uvp, 0, nullptr);
01018     vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
01019                           VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
01020                           &before_barrier_directions, 0, nullptr);
01021
01022     vkCmdUpdateBuffer(command_buffers[image_index],
01023                       globalUBOBuffer[image_index].getBuffer(), 0,
01024                       sizeof(GlobalUBO), &globalUBO);
01025     vkCmdUpdateBuffer(command_buffers[image_index],
01026                       sceneUBOBuffer[image_index].getBuffer(), 0,
01027                       sizeof(SceneUBO), &sceneUBO);
01028
01029     VkBufferMemoryBarrier after_barrier_uvp{};
01030     after_barrier_uvp.pNext = nullptr;
01031     after_barrier_uvp.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01032     after_barrier_uvp.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01033     after_barrier_uvp.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
01034     after_barrier_uvp.buffer = globalUBOBuffer[image_index].getBuffer();
01035     after_barrier_uvp.offset = 0;
01036     after_barrier_uvp.size = sizeof(GlobalUBO);
01037     after_barrier_uvp.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01038     after_barrier_uvp.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01039
01040     VkBufferMemoryBarrier after_barrier_directions{};
01041     after_barrier_directions.pNext = nullptr;
01042     after_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01043     after_barrier_directions.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01044     after_barrier_directions.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
01045     after_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
01046     after_barrier_directions.offset = 0;
01047     after_barrier_directions.size = sizeof(SceneUBO);
01048     after_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01049     after_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01050
01051     vkCmdPipelineBarrier(command_buffers[image_index],
01052                           VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
01053                           nullptr, 1, &after_barrier_uvp, 0, nullptr);
01054     vkCmdPipelineBarrier(command_buffers[image_index],
01055                           VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
01056                           nullptr, 1, &after_barrier_directions, 0, nullptr);
01057 }
01058
01059 void VulkanRenderer::update_raytracing_descriptor_set(uint32_t image_index) {
01060     VkWriteDescriptorSetAccelerationStructureKHR
01061         descriptor_set_acceleration_structure{};
01062     descriptor_set_acceleration_structure.sType =
01063         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
01064     descriptor_set_acceleration_structure.pNext = nullptr;
01065     descriptor_set_acceleration_structure.accelerationStructureCount = 1;
01066     VkAccelerationStructureKHR& tlasAS = asManager.getTLAS();
01067     descriptor_set_acceleration_structure.pAccelerationStructures = &tlasAS;
01068
01069     VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
01070     write_descriptor_set_acceleration_structure.sType =
01071         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
01072     write_descriptor_set_acceleration_structure.pNext =
01073         &descriptor_set_acceleration_structure;
01074     write_descriptor_set_acceleration_structure.dstSet =

```

```
01075     raytracingDescriptorSet[image_index];
01076     write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
01077     write_descriptor_set_acceleration_structure.dstArrayElement = 0;
01078     write_descriptor_set_acceleration_structure.descriptorCount = 1;
01079     write_descriptor_set_acceleration_structure.descriptorType =
01080         VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
01081     write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
01082     write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
01083     write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;
01084
01085     VkDescriptorBufferInfo object_description_buffer_info{};
01086     object_description_buffer_info.buffer = objectDescriptionBuffer.getBuffer();
01087     object_description_buffer_info.offset = 0;
01088     object_description_buffer_info.range = VK_WHOLE_SIZE;
01089
01090     VkWriteDescriptorSet object_description_buffer_write{};
01091     object_description_buffer_write.sType =
01092         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
01093     object_description_buffer_write.dstSet =
01094         sharedRenderDescriptorSet[image_index];
01095     object_description_buffer_write.descriptorType =
01096         VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
01097     object_description_buffer_write.dstBinding = OBJECT_DESCRIPTION_BINDING;
01098     object_description_buffer_write.pBufferInfo = &object_description_buffer_info;
01099     object_description_buffer_write.descriptorCount = 1;
01100
01101     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
01102         write_descriptor_set_acceleration_structure,
01103         object_description_buffer_write};
01104
01105     vkUpdateDescriptorSets(device->getLogicalDevice(),
01106                             static_cast<uint32_t>(write_descriptor_sets.size()),
01107                             write_descriptor_sets.data(), 0, nullptr);
01108 }
01109
01110 void VulkanRenderer::record_commands(uint32_t image_index) {
01111     Texture& renderResult = rasterizer.getOffscreenTexture(image_index);
01112     VulkanImage& vulkanImage = renderResult.getVulkanImage();
01113
01114     GUIRendererSharedVars& guiRendererSharedVars =
01115         gui->getGuiRendererSharedVars();
01116     if (guiRendererSharedVars.raytracing) {
01117         std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],
01118                                             raytracingDescriptorSet[image_index]};
01119         raytracingStage.recordCommands(command_buffers[image_index],
01120                                         &vulkanSwapChain, sets);
01121
01122     } else if (guiRendererSharedVars.pathTracing) {
01123         std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],
01124                                             raytracingDescriptorSet[image_index]};
01125
01126         pathTracing.recordCommands(command_buffers[image_index], image_index,
01127                                     vulkanImage, &vulkanSwapChain, sets);
01128
01129     } else {
01130         std::vector<VkDescriptorSet> descriptorSets = {
01131             sharedRenderDescriptorSet[image_index]};
01132
01133         rasterizer.recordCommands(command_buffers[image_index], image_index, scene,
01134                                     descriptorSets);
01135     }
01136
01137     vulkanImage.transitionImageLayout(
01138         command_buffers[image_index], VK_IMAGE_LAYOUT_GENERAL,
01139         VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);
01140
01141     std::vector<VkDescriptorSet> descriptorSets = {
01142         post_descriptor_set[image_index]};
01143     postStage.recordCommands(command_buffers[image_index], image_index,
01144                               descriptorSets);
01145
01146     vulkanImage.transitionImageLayout(
01147         command_buffers[image_index], VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
01148         VK_IMAGE_LAYOUT_GENERAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);
01149 }
01150
01151 bool VulkanRenderer::checkChangedFramebufferSize() {
01152     if (window->framebuffer_size_has_changed()) {
01153         vkDeviceWaitIdle(device->getLogicalDevice());
01154         vkQueueWaitIdle(device->getGraphicsQueue());
01155
01156         vulkanSwapChain.cleanUp();
01157         vulkanSwapChain.initVulkanContext(device.get(), window, surface);
01158
01159         std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
01160             sharedRenderDescriptorSetLayout};
01161         rasterizer.cleanUp();
01162 }
```

```

01162     rasterizer.init(device.get(), &vulkanSwapChain, descriptor_set_layouts,
01163             graphics_command_pool);
01164
01165 // all post
01166 std::vector<VkDescriptorSetLayout> descriptorSets = {
01167     post_descriptor_set_layout;
01168 postStage.cleanUp();
01169 postStage.init(device.get(), &vulkanSwapChain, descriptorSets);
01170
01171 gui->cleanUp();
01172 gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
01173                                 postStage.getRenderPass(),
01174                                 graphics_command_pool);
01175
01176 current_frame = 0;
01177
01178 updatePostDescriptorSets();
01179 updateRaytracingDescriptorSets();
01180
01181 window->reset_framebuffer_has_changed();
01182
01183 return true;
01184 }
01185
01186 return false;
01187 }
01188
01189 void VulkanRenderer::cleanUp() {
01190     cleanUpUBOs();
01191
01192     rasterizer.cleanUp();
01193     raytracingStage.cleanUp();
01194     postStage.cleanUp();
01195     pathTracing.cleanUp();
01196
01197     objectDescriptionBuffer.cleanUp();
01198     asManager.cleanUp();
01199
01200     vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01201         raytracingDescriptorSetLayout, nullptr);
01202     vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01203         post_descriptor_set_layout, nullptr);
01204     vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01205         sharedRenderDescriptorSetLayout, nullptr);
01206     vkDestroyDescriptorPool(device->getLogicalDevice(), post_descriptor_pool,
01207         nullptr);
01208     vkDestroyDescriptorPool(device->getLogicalDevice(),
01209         descriptorPoolSharedRenderStages, nullptr);
01210     vkDestroyDescriptorPool(device->getLogicalDevice(), raytracingDescriptorPool,
01211         nullptr);
01212
01213     vkFreeCommandBuffers(device->getLogicalDevice(), graphics_command_pool,
01214         static_cast<uint32_t>(command_buffers.size()),
01215         command_buffers.data());
01216
01217     cleanUpCommandPools();
01218
01219     cleanUpSync();
01220
01221     vulkanSwapChain.cleanUp();
01222     vkDestroySurfaceKHR(instance.getVulkanInstance(), surface, nullptr);
01223     allocator.cleanUp();
01224     device->cleanUp();
01225     debug::freeDebugCallback(instance.getVulkanInstance());
01226     instance.cleanUp();
01227 }
01228
01229 VulkanRenderer::~VulkanRenderer() {}

```

## 6.185 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/← Camera.cpp File Reference

### 6.186 Camera.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Camera.hpp"
00002

```

```
00003 Camera::Camera()
00004     :
00005
00006     position(glm::vec3(0.0f, 100.0f, -80.0f)),
00007     front(glm::vec3(0.0f, 0.0f, -1.0f)),
00008     world_up(glm::vec3(0.0f, 1.0f, 0.0f)),
00009     right(glm::normalize(glm::cross(front, world_up))),
00010     up(glm::normalize(glm::cross(right, front))),
00011     yaw(80.0f),
00012     pitch(-40.0f),
00013     movement_speed(200.0f),
00014     turn_speed(0.25f),
00015     near_plane(0.1f),
00016     far_plane(4000.0f),
00017     fov(45.0f)
00018
00019 {}
00020
00021 void Camera::key_control(bool* keys, float delta_time) {
00022     float velocity = movement_speed * delta_time;
00023
00024     if (keys[GLFW_KEY_W]) {
00025         position += front * velocity;
00026     }
00027
00028     if (keys[GLFW_KEY_D]) {
00029         position += right * velocity;
00030     }
00031
00032     if (keys[GLFW_KEY_A]) {
00033         position += -right * velocity;
00034     }
00035
00036     if (keys[GLFW_KEY_S]) {
00037         position += -front * velocity;
00038     }
00039
00040     if (keys[GLFW_KEY_Q]) {
00041         yaw += -velocity;
00042     }
00043
00044     if (keys[GLFW_KEY_E]) {
00045         yaw += velocity;
00046     }
00047 }
00048
00049 void Camera::mouse_control(float x_change, float y_change) {
00050     // here we only want to support views 90 degrees to each side
00051     // again choose turn speed well in respect to its ordinal scale
00052     x_change *= turn_speed;
00053     y_change *= turn_speed;
00054
00055     yaw += x_change;
00056     pitch += y_change;
00057
00058     if (pitch > 89.0f) {
00059         pitch = 89.0f;
00060     }
00061
00062     if (pitch < -89.0f) {
00063         pitch = -89.0f;
00064     }
00065
00066     // by changing the rotations you need to update all parameters
00067     // for we retrieve them later for further calculations!
00068     update();
00069 }
00070
00071 void Camera::set_near_plane(float near_plane) { this->near_plane = near_plane; }
00072
00073 void Camera::set_far_plane(float far_plane) { this->far_plane = far_plane; }
00074
00075 void Camera::set_fov(float fov) { this->fov = fov; }
00076
00077 void Camera::set_camera_position(glm::vec3 new_camera_position) {
00078     this->position = new_camera_position;
00079 }
00080
00081 glm::mat4 Camera::calculate_viewmatrix() {
00082     // very necessary for further calc
00083     return glm::lookAt(position, position + front, up);
00084 }
00085
00086 Camera::~Camera() {}
00087
00088 void Camera::update() {
00089     //
```

```

https://learnopengl.com/Getting-started/Camera?fbclid=IwAR1WEr4jt6IyWC52s_WKYHtaFoeug37pG5YqbDPifgn5F1UXPbUjWbJWiQ
00090 // thats a bit tricky; have a look to link above if there a questions :)
00091 // but simple geometrical analysis
00092 // consider yaw you are turnig to the side; pitch as you move the head forward
00093 // and back; roll rotations around z-axis will make you dizzy :)) notice that
00094 // to roll will not chnge my front vector
00095 front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
00096 front.y = sin(glm::radians(pitch));
00097 front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
00098 front = glm::normalize(front);
00099
00100 // retrieve the right vector with some world_up
00101 right = glm::normalize(glm::cross(front, world_up));
00102
00103 // but this means the up vector must again be calculated with right vector
00104 // calculated!!!
00105 up = glm::normalize(glm::cross(right, front));
00106 }

```

## 6.187 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/scene/← Mesh.cpp File Reference

### 6.188 Mesh.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Mesh.hpp"
00002
00003 #include <cstring>
00004 #include <memory>
00005
00006 #include "VulkanBuffer.hpp"
00007
00008 Mesh::Mesh() {}
00009
00010 void Mesh::cleanUp() {
00011     vertexBuffer.cleanUp();
00012     indexBuffer.cleanUp();
00013     objectDescriptionBuffer.cleanUp();
00014     materialIdsBuffer.cleanUp();
00015     materialsBuffer.cleanUp();
00016 }
00017
00018 Mesh::Mesh(VulkanDevice* device, VkQueue transfer_queue,
00019             VkCommandPool transfer_command_pool, std::vector<Vertex>& vertices,
00020             std::vector<uint32_t>& indices,
00021             std::vector<unsigned int>& materialIndex,
00022             std::vector<ObjMaterial>& materials) {
00023     // glm uses column major matrices so transpose it for Vulkan want row major
00024     // here
00025     glm::mat4 transpose_transform = glm::transpose(glm::mat4(1.0f));
00026     VkTransformMatrixKHR out_matrix;
00027     std::memcpy(&out_matrix, &transpose_transform, sizeof(VkTransformMatrixKHR));
00028
00029     index_count = static_cast<uint32_t>(indices.size());
00030     vertex_count = static_cast<uint32_t>(vertices.size());
00031     this->device = device;
00032     object_description = ObjectDescription{};
00033     createVertexBuffer(transfer_queue, transfer_command_pool, vertices);
00034     createIndexBuffer(transfer_queue, transfer_command_pool, indices);
00035     createMaterialIDBuffer(transfer_queue, transfer_command_pool, materialIndex);
00036     createMaterialBuffer(transfer_queue, transfer_command_pool, materials);
00037
00038     VkBufferDeviceAddressInfo vertex_info{};
00039     vertex_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00040     vertex_info.buffer = vertexBuffer.getBuffer();
00041
00042     VkBufferDeviceAddressInfo index_info{};
00043     index_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00044     index_info.buffer = indexBuffer.getBuffer();
00045
00046     VkBufferDeviceAddressInfo material_index_info{};
00047     material_index_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00048     material_index_info.buffer = materialIdsBuffer.getBuffer();
00049
00050     VkBufferDeviceAddressInfo material_info{};
00051     material_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
00052     material_info.buffer = materialsBuffer.getBuffer();

```

```

00053
00054     object_description.index_address =
00055         vkGetBufferDeviceAddress(device->getLogicalDevice(), &index_info);
00056     object_description.vertex_address =
00057         vkGetBufferDeviceAddress(device->getLogicalDevice(), &vertex_info);
00058     object_description.material_index_address = vkGetBufferDeviceAddress(
00059         device->getLogicalDevice(), &material_index_info);
00060     object_description.material_address =
00061         vkGetBufferDeviceAddress(device->getLogicalDevice(), &material_info);
00062
00063     model = glm::mat4(1.0f);
00064 }
00065
00066 void Mesh::setModel(glm::mat4 new_model) { model = new_model; }
00067
00068 Mesh::~Mesh() {}
00069
00070 void Mesh::createVertexBuffer(VkQueue transfer_queue,
00071                                 VkCommandPool transfer_command_pool,
00072                                 std::vector<Vertex>& vertices) {
00073     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00074         device, transfer_command_pool, vertexBuffer,
00075         VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT |
00076             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00077             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00078             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00079         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00080             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00081         vertices);
00082 }
00083
00084 void Mesh::createIndexBuffer(VkQueue transfer_queue,
00085                                 VkCommandPool transfer_command_pool,
00086                                 std::vector<uint32_t>& indices) {
00087     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00088         device, transfer_command_pool, indexBuffer,
00089         VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
00090             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00091             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00092             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00093         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00094             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00095         indices);
00096 }
00097
00098 void Mesh::createMaterialIDBuffer(VkQueue transfer_queue,
00099                                 VkCommandPool transfer_command_pool,
00100                                 std::vector<unsigned int>& materialIndex) {
00101     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00102         device, transfer_command_pool, materialIdsBuffer,
00103         VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
00104             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00105             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00106             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00107         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00108             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00109         materialIndex);
00110 }
00111
00112 void Mesh::createMaterialBuffer(VkQueue transfer_queue,
00113                                 VkCommandPool transfer_command_pool,
00114                                 std::vector<ObjMaterial>& materials) {
00115     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00116         device, transfer_command_pool, materialsBuffer,
00117         VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
00118             VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00119             VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
00120             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00121         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00122             VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00123         materials);
00124 }

```

## 6.189 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/scene/← Model.cpp File Reference

### 6.190 Model.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Model.hpp"
00002
00003 Model::Model() {}
00004
00005 Model::Model(VulkanDevice* device) { this->device = device; }
00006
00007 void Model::cleanUp() {
00008     for (Texture texture : modelTextures) {
00009         texture.cleanUp();
00010     }
00011
00012     for (VkSampler texture_sampler : modelTextureSamplers) {
00013         vkDestroySampler(device->getLogicalDevice(), texture_sampler, nullptr);
00014     }
00015
00016     mesh.cleanUp();
00017 }
00018
00019 void Model::add_new_mesh(VulkanDevice* device, VkQueue transfer_queue,
00020                             VkCommandPool command_pool,
00021                             std::vector<Vertex>& vertices,
00022                             std::vector<unsigned int>& indices,
00023                             std::vector<unsigned int>& materialIndex,
00024                             std::vector<ObjMaterial>& materials) {
00025     this->mesh = Mesh(device, transfer_queue, command_pool, vertices, indices,
00026                         materialIndex, materials);
00027 }
00028
00029 void Model::set_model(glm::mat4 model) { this->model = model; }
00030
00031 void Model::addTexture(Texture newTexture) {
00032     modelTextures.push_back(newTexture);
00033     addSampler(newTexture);
00034 }
00035
00036 uint32_t Model::getPrimitiveCount() {
00037     /*uint32_t number_of_indices = 0;
00038
00039     for (Mesh mesh : meshes) {
00040         number_of_indices += mesh.get_index_count();
00041     }
00042
00043     return number_of_indices / 3; */
00044     return mesh.getIndexCount() / 3;
00045 }
00046
00047 }
00048
00049 Model::~Model() {}
00050
00051 void Model::addSampler(Texture newTexture) {
00052     VkSampler newSampler;
00053     // sampler create info
00054     VkSamplerCreateInfo sampler_create_info{};
00055     sampler_create_info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
00056     sampler_create_info.magFilter = VK_FILTER_LINEAR;
00057     sampler_create_info.minFilter = VK_FILTER_LINEAR;
00058     sampler_create_info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00059     sampler_create_info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00060     sampler_create_info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00061     sampler_create_info.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
00062     sampler_create_info.unnormalizedCoordinates = VK_FALSE;
00063     sampler_create_info.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
00064     sampler_create_info.mipLodBias = 0.0f;
00065     sampler_create_info.minLod = 0.0f;
00066     sampler_create_info.maxLod = newTexture.getMipLevel();
00067     sampler_create_info.anisotropyEnable = VK_TRUE;
00068     sampler_create_info.maxAnisotropy = 16; // max anisotropy sample level
00069
00070     VkResult result = vkCreateSampler(device->getLogicalDevice(),
00071                                         &sampler_create_info, nullptr, &newSampler);
00072     ASSERT_VULKAN(result, "Failed to create a texture sampler!")
00073
00074     modelTextureSamplers.push_back(newSampler);
00075 }

```

## 6.191 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/scene/ObjLoader.cpp File Reference

### 6.192 ObjLoader.cpp

[Go to the documentation of this file.](#)

```

00001 #include "ObjLoader.hpp"
00002 #define TINYOBJLOADER_IMPLEMENTATION
00003 #include <tiny_obj_loader.h>
00004
00005 #include "File.hpp"
00006
00007 ObjLoader::ObjLoader(VulkanDevice* device, VkQueue transfer_queue,
00008                         VkCommandPool command_pool) {
00009     this->device = device;
00010     this->transfer_queue = transfer_queue;
00011     this->command_pool = command_pool;
00012 }
00013
00014 std::shared_ptr<Model> ObjLoader::loadModel(const std::string& modelFile) {
00015     // the model we want to load
00016     std::shared_ptr<Model> new_model = std::make_shared<Model>(device);
00017
00018     // first load textures from model
00019     std::vector<std::string> textureNames = loadTexturesAndMaterials(modelFile);
00020     std::vector<int> matToTex(textureNames.size());
00021
00022     // now that we have the names lets create the vulkan side of textures
00023     for (size_t i = 0; i < textureNames.size(); i++) {
00024         // If material had no texture, set '0' to indicate no texture, texture 0
00025         // will be reserved for a default texture
00026         if (!textureNames[i].empty()) {
00027             // Otherwise, create texture and set value to index of new texture
00028             Texture texture;
00029             texture.createFromFile(device, command_pool, textureNames[i]);
00030             new_model->addTexture(texture);
00031             matToTex[i] = new_model->getTextureCount();
00032
00033         } else {
00034             matToTex[i] = 0;
00035         }
00036     }
00037
00038     loadVertices(modelFile);
00039
00040     new_model->add_new_mesh(device, transfer_queue, command_pool, vertices,
00041                             indices, materialIndex, this->materials);
00042
00043     return new_model;
00044 }
00045
00046 std::vector<std::string> ObjLoader::loadTexturesAndMaterials(
00047     const std::string& modelFile) {
00048     tinyobj::ObjReaderConfig reader_config;
00049     tinyobj::ObjReader reader;
00050
00051     if (!reader.ParseFromFile(modelFile, reader_config)) {
00052         if (!reader.Error().empty()) {
00053             std::cerr << "TinyObjReader: " << reader.Error();
00054         }
00055         exit(EXIT_FAILURE);
00056     }
00057
00058     if (!reader.Warning().empty()) {
00059         std::cout << "TinyObjReader: " << reader.Warning();
00060     }
00061
00062     auto& tol_materials = reader.GetMaterials();
00063     textures.reserve(tol_materials.size());
00064
00065     int texture_id = 0;
00066
00067     // we now iterate over all materials to get diffuse textures
00068     for (size_t i = 0; i < tol_materials.size(); i++) {
00069         const tinyobj::material_t* mp = &tol_materials[i];
00070         ObjMaterial material{};
00071         material.ambient =
00072             glm::vec3(mp->ambient[0], mp->ambient[1], mp->ambient[2]);
00073         material.diffuse =
00074             glm::vec3(mp->diffuse[0], mp->diffuse[1], mp->diffuse[2]);
00075         material.specular =

```

```

00076     glm::vec3(mp->specular[0], mp->specular[1], mp->specular[2]);
00077     material.emission =
00078         glm::vec3(mp->emission[0], mp->emission[1], mp->emission[2]);
00079     material.transmittance = glm::vec3(
00080         mp->transmittance[0], mp->transmittance[1], mp->transmittance[2]);
00081     material.dissolve = mp->dissolve;
00082     material.ior = mp->ior;
00083     material.shininess = mp->shininess;
00084     material.illum = mp->illum;
00085
00086     if (mp->diffuse_texname.length() > 0) {
00087         std::string relative_texture_filename = mp->diffuse_texname;
00088         File model_file(modelFile);
00089         std::string texture_filename =
00090             model_file.getBaseDir() + "/textures/" + relative_texture_filename;
00091
00092         textures.push_back(texture_filename);
00093         material.textureID = texture_id;
00094         texture_id++;
00095
00096     } else {
00097         material.textureID = 0;
00098         textures.push_back("");
00099     }
00100
00101     materials.push_back(material);
00102 }
00103
00104 // for the case no .mtl file is given place some random standard material ...
00105 if (tol_materials.empty()) {
00106     materials.emplace_back(ObjMaterial());
00107 }
00108
00109 return textures;
00110 }
00111
00112 void ObjLoader::loadVertices(const std::string& fileName) {
00113     tinyobj::ObjReaderConfig reader_config;
00114     // reader_config.mtl_search_path = ""; // Path to material files
00115
00116     tinyobj::ObjReader reader;
00117
00118     if (!reader.ParseFromFile(fileName, reader_config)) {
00119         if (!reader.Error().empty()) {
00120             std::cerr << "TinyObjReader: " << reader.Error();
00121         }
00122         exit(EXIT_FAILURE);
00123     }
00124
00125     if (!reader.Warning().empty()) {
00126         std::cout << "TinyObjReader: " << reader.Warning();
00127     }
00128
00129     auto& attrib = reader.GetAttrib();
00130     auto& shapes = reader.GetShapes();
00131     auto& materials = reader.GetMaterials();
00132
00133     std::unordered_map<Vertex, uint32_t> vertices_map{};
00134
00135     // Loop over shapes
00136     for (size_t s = 0; s < shapes.size(); s++) {
00137         // prepare for enlargement
00138         vertices.reserve(shapes[s].mesh.indices.size() + vertices.size());
00139         indices.reserve(shapes[s].mesh.indices.size() + indices.size());
00140
00141         // Loop over faces(polygon)
00142         size_t index_offset = 0;
00143         for (size_t f = 0; f < shapes[s].mesh.num_face_vertices.size(); f++) {
00144             size_t fv = size_t(shapes[s].mesh.num_face_vertices[f]);
00145
00146             // Loop over vertices in the face.
00147             for (size_t v = 0; v < fv; v++) {
00148                 // access to vertex
00149                 tinyobj::index_t idx = shapes[s].mesh.indices[index_offset + v];
00150                 tinyobj::real_t vx = attrib.vertices[3 * size_t(idx.vertex_index) + 0];
00151                 tinyobj::real_t vy = attrib.vertices[3 * size_t(idx.vertex_index) + 1];
00152                 tinyobj::real_t vz = attrib.vertices[3 * size_t(idx.vertex_index) + 2];
00153                 glm::vec3 pos = {vx, vy, vz};
00154
00155                 glm::vec3 normals(0.0f);
00156                 // Check if 'normal_index' is zero or positive. negative = no normal
00157                 // data
00158                 if (idx.normal_index >= 0 && !attrib.normals.empty()) {
00159                     tinyobj::real_t nx = attrib.normals[3 * size_t(idx.normal_index) + 0];
00160                     tinyobj::real_t ny = attrib.normals[3 * size_t(idx.normal_index) + 1];
00161                     tinyobj::real_t nz = attrib.normals[3 * size_t(idx.normal_index) + 2];
00162                     normals = glm::vec3(nx, ny, nz);
00163             }
00164         }
00165     }
00166 }
```

```

00163         }
00164
00165         glm::vec3 color(-1.f);
00166         if (!attrib.colors.empty()) {
00167             tinyobj::real_t red = attrib.colors[3 * size_t(idx.vertex_index) + 0];
00168             tinyobj::real_t green =
00169                 attrib.colors[3 * size_t(idx.vertex_index) + 1];
00170             tinyobj::real_t blue =
00171                 attrib.colors[3 * size_t(idx.vertex_index) + 2];
00172             color = glm::vec3(red, green, blue);
00173         }
00174
00175         glm::vec2 tex_coords(0.0f);
00176         // Check if 'texcoord_index' is zero or positive. negative = no texcoord
00177         // data
00178         if (idx.texcoord_index >= 0 && !attrib.texcoords.empty()) {
00179             tinyobj::real_t tx =
00180                 attrib.texcoords[2 * size_t(idx.texcoord_index) + 0];
00181             // flip y coordinate !!
00182             tinyobj::real_t ty =
00183                 1.f - attrib.texcoords[2 * size_t(idx.texcoord_index) + 1];
00184             tex_coords = glm::vec2(tx, ty);
00185         }
00186
00187         Vertex vert{pos, normals, color, tex_coords};
00188
00189         if (vertices_map.count(vert) == 0) {
00190             vertices_map[vert] = vertices.size();
00191             vertices.push_back(vert);
00192         }
00193
00194         indices.push_back(vertices_map[vert]);
00195     }
00196
00197     index_offset += fv;
00198
00199     // per-face material; face usually is triangle
00200     // matToTex[shapes[s].mesh.material_ids[f]]
00201     materialIndex.push_back(shapes[s].mesh.material_ids[f]);
00202 }
00203 }
00204
00205 // precompute normals if no provided
00206 if (attrib.normals.empty()) {
00207     for (size_t i = 0; i < indices.size(); i += 3) {
00208         Vertex& v0 = vertices[indices[i + 0]];
00209         Vertex& v1 = vertices[indices[i + 1]];
00210         Vertex& v2 = vertices[indices[i + 2]];
00211
00212         glm::vec3 n =
00213             glm::normalize(glm::cross((v1.pos - v0.pos), (v2.pos - v0.pos)));
00214         v0.normal = n;
00215         v1.normal = n;
00216         v2.normal = n;
00217     }
00218 }
00219 }
```

## 6.193 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/scene/Scene.cpp File Reference

### 6.194 Scene.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Scene.hpp"
00002
00003 Scene::Scene() {}
00004
00005 void Scene::update_user_input(GUI* gui) {
00006     guiSceneSharedVars = gui->getGuiSceneSharedVars();
00007 }
00008
00009 void Scene::loadModel(VulkanDevice* device, VkCommandPool commandPool) {
00010     ObjLoader obj_loader(device, device->getGraphicsQueue(), commandPool);
00011
00012     std::string modelFileName = sceneConfig::getModelFile();
00013     std::shared_ptr<Model> new_model = obj_loader.loadModel(modelFileName);
```

```

00014     add_model(new_model);
00015
00016     glm::mat4 modelMatrix = sceneConfig::getModelMatrix();
00017
00018     update_model_matrix(modelMatrix, 0);
00019
00020 }
00021
00022 void Scene::add_model(std::shared_ptr<Model> model) {
00023     model_list.push_back(model);
00024     object_descriptions.push_back(model->getObjectDescription());
00025 }
00026
00027 void Scene::add_object_description(ObjectDescription object_description) {
00028     object_descriptions.push_back(object_description);
00029 }
00030
00031 void Scene::update_model_matrix(glm::mat4 model_matrix, int model_id) {
00032     if (model_id >= static_cast<int32_t>(getModelCount()) || model_id < 0) {
00033         throw std::runtime_error("Wrong model id value!");
00034     }
00035
00036     model_list[model_id]->set_model(model_matrix);
00037 }
00038
00039 void Scene::cleanUp() {
00040     for (std::shared_ptr<Model> model : model_list) {
00041         model->cleanUp();
00042     }
00043 }
00044
00045 uint32_t Scene::getNumberMeshes() {
00046     uint32_t number_of_meshes = 0;
00047
00048     for (std::shared_ptr<Model> mesh_model : model_list) {
00049         number_of_meshes += static_cast<uint32_t>(mesh_model->getMeshCount());
00050     }
00051
00052     return number_of_meshes;
00053 }
00054
00055 Scene::~Scene() {}

```

## 6.195 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/SceneConfig.cpp File Reference

### Namespaces

- namespace `sceneConfig`

### Functions

- `std::string sceneConfig::getModelFile ()`
- `glm::mat4 sceneConfig::getModelMatrix ()`

## 6.196 SceneConfig.cpp

[Go to the documentation of this file.](#)

```

00001 #include "SceneConfig.hpp"
00002 #include "VulkanRendererConfig.hpp"
00003
00004 #include <filesystem>
00005 //#define SULO_MODE 1
00006
00007 namespace sceneConfig {
00008
00009 std::string getModelFile() {
0010     std::stringstream modelFile;

```

```

00011     std::filesystem::path cwd = std::filesystem::current_path();
00012     modelFile << cwd.string();
00013     modelFile << RELATIVE_RESOURCE_PATH;
00014
00015 #if NDEBUG
00016     modelFile << "Models/crytek-sponza/";
00017     modelFile << "sponza_triang.obj";
00018
00019 #else
00020 #ifdef SULO_MODE
00021     modelFile << "Model/Sulo/WolfStahl/";
00022 //modelfile << "Wolf-Stahl.obj";
00023     modelFile << "SuloLongDongLampe_v2.obj";
00024 #else
00025     modelFile << "Models/VikingRoom/";
00026     modelFile << "viking_room.obj";
00027 #endif
00028 #endif
00029
00030     return modelFile.str();
00031 // std::string modelFile =
00032 // "Models/crytek-sponza/sponza_triang.obj"; std::string modelFile
00033 // = "Models/Dinosaurs/dinosaurs.obj"; std::string modelFile =
00034 // "Models/Pillum/PillumPainting_export.obj"; std::string modelFile
00035 // = "Models/sibenik/sibenik.obj"; std::string modelFile =
00036 // "Models/sportsCar/sportsCar.obj"; std::string modelFile =
00037 // "Models/StanfordDragon/dragon.obj"; std::string modelFile =
00038 // "Models/CornellBox/CornellBox-Sphere.obj"; std::string
00039 // "Models/bunny/bunny.obj"; std::string modelFile =
00040 // "Models/buddha/buddha.obj"; std::string modelFile =
00041 // "Models/bmw/bmw.obj"; std::string modelFile =
00042 // "Models/testScene.obj"; std::string modelFile =
00043 // "Models/San_Miguel/san-miguel-low-poly.obj";
00044 }
00045
00046 glm::mat4 getModelMatrix() {
00047     glm::mat4 modelMatrix(1.0f);
00048
00049 #if NDEBUG
00050
00051 // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00052 // -50.0f));
00053 modelMatrix = glm::scale(modelMatrix, glm::vec3(1.0f, 1.0f, 1.0f));
00054 /*dragon_model = glm::rotate(dragon_model, glm::radians(-90.f),
00055 glm::vec3(1.0f, 0.0f, 0.0f)); dragon_model = glm::rotate(dragon_model,
00056 glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));*/
00057
00058 #else
00059
00060 // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00061 // -50.0f));
00062 #if SULO_MODE
00063     modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00064 #else
00065     modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00066     modelMatrix = glm::rotate(modelMatrix, glm::radians(-90.f),
00067                             glm::vec3(1.0f, 0.0f, 0.0f));
00068     modelMatrix =
00069         glm::rotate(modelMatrix, glm::radians(90.f), glm::vec3(0.0f, 0.0f, 1.0f));
00070 #endif
00071
00072 #endif
00073
00074     return modelMatrix;
00075 }
00076
00077 } // namespace sceneConfig

```

## 6.197 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/scene/← Texture.cpp File Reference

### 6.198 Texture.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Texture.hpp"
00002
00003 #include <cmath>

```

```

00004 #include <stdexcept>
00005
00006 Texture::Texture() {}
00007
00008 void Texture::createFromFile(VulkanDevice* device, VkCommandPool commandPool,
00009                                     const std::string& fileName) {
0010     int width, height;
0011     VkDeviceSize size;
0012     stbi_uc* image_data = loadTextureData(fileName, &width, &height, &size);
0013
0014     mip_levels =
0015         static_cast<uint32_t>(std::floor(std::log2(std::max(width, height)))) + 1;
0016
0017     // create staging buffer to hold loaded data, ready to copy to device
0018     VulkanBuffer stagingBuffer;
0019     stagingBuffer.create(device, size, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
0020                           VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
0021                           VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);
0022
0023     // copy image data to staging buffer
0024     void* data;
0025     vkMapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory(), 0,
0026                  size, 0, &data);
0027     memcpy(data, image_data, static_cast<size_t>(size));
0028     vkUnmapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory());
0029
0030     // free original image data
0031     stbi_image_free(image_data);
0032
0033     createImage(device, width, height, mip_levels, VK_FORMAT_R8G8B8A8_UNORM,
0034                 VK_IMAGE_TILING_OPTIMAL,
0035                 VK_IMAGE_USAGE_TRANSFER_SRC_BIT |
0036                 VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
0037                 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
0038
0039     // copy data to image
0040     // transition image to be DST for copy operation
0041     vulkanImage.transitionImageLayout(
0042         device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
0043         VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
0044         VK_IMAGE_ASPECT_COLOR_BIT, mip_levels);
0045
0046     // copy data to image
0047     vulkanBufferManager.copyImageBuffer(
0048         device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
0049         stagingBuffer.getBuffer(), vulkanImage.getImage(), width, height);
0050
0051     // generate mipmaps
0052     generateMipMaps(device->getPhysicalDevice(), device->getLogicalDevice(),
0053                       commandPool, device->getGraphicsQueue(),
0054                       vulkanImage.getImage(), VK_FORMAT_R8G8B8A8_SRGB, width,
0055                       height, mip_levels);
0056
0057     stagingBuffer.cleanUp();
0058
0059     createImageView(device, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_COLOR_BIT,
0060                     mip_levels);
0061 }
0062
0063 void Texture::setImage(VkImage image) { vulkanImage.setImage(image); }
0064
0065 void Texture::setImageView(VkImageView imageView) {
0066     vulkanImageView.setImageView(imageView);
0067 }
0068
0069 void Texture::createImage(VulkanDevice* device, uint32_t width, uint32_t height,
0070                           uint32_t mip_levels, VkFormat format,
0071                           VkImageTiling tiling, VkImageUsageFlags use_flags,
0072                           VkMemoryPropertyFlags prop_flags) {
0073     vulkanImage.create(device, width, height, mip_levels, format, tiling,
0074                         use_flags, prop_flags);
0075 }
0076
0077 void Texture::createImageView(VulkanDevice* device, VkFormat format,
0078                               VkImageAspectFlags aspect_flags,
0079                               uint32_t mip_levels) {
0080     vulkanImageView.create(device, vulkanImage.getImage(), format, aspect_flags,
0081                           mip_levels);
0082 }
0083
0084 void Texture::cleanUp() {
0085     vulkanImageView.cleanUp();
0086     vulkanImage.cleanUp();
0087 }
0088
0089 Texture::~Texture() {}
0090

```

```
00091 stbi_uc* Texture::loadTextureData(const std::string& file_name, int* width,
00092                                     int* height, VkDeviceSize* image_size) {
00093     // number of channels image uses
00094     int channels;
00095     // load pixel data for image
00096     // std::string file_loc = "../Resources/Textures/" + file_name;
00097     stbi_uc* image =
00098         stbi_load(file_name.c_str(), width, height, &channels, STBI_rgb_alpha);
00099
00100     if (!image) {
00101         throw std::runtime_error("Failed to load a texture file! (" + file_name +
00102                                 ")");
00103     }
00104
00105     // calculate image size using given and known data
00106     *image_size = *width * *height * 4;
00107
00108     return image;
00109 }
00110
00111 void Texture::generateMipMaps(VkPhysicalDevice physical_device, VkDevice device,
00112                                  VkCommandPool command_pool, VkQueue queue,
00113                                  VkImage image, VkFormat image_format,
00114                                  int32_t width, int32_t height,
00115                                  uint32_t mip_levels) {
00116     // Check if image format supports linear blitting
00117     VkFormatProperties formatProperties;
00118     vkGetPhysicalDeviceFormatProperties(physical_device, image_format,
00119                                         &formatProperties);
00120
00121     if (!(formatProperties.optimalTilingFeatures &
00122           VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT)) {
00123         throw std::runtime_error(
00124             "Texture image format does not support linear blitting!");
00125     }
00126
00127     VkCommandBuffer command_buffer =
00128         commandBufferManager.beginCommandBuffer(device, command_pool);
00129
00130     VkImageMemoryBarrier barrier{};
00131     barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00132     barrier.image = image;
00133     barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
00134     barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
00135     barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00136     barrier.subresourceRange.baseArrayLayer = 0;
00137     barrier.subresourceRange.layerCount = 1;
00138     barrier.subresourceRange.levelCount = 1;
00139
00140     // TEMP VARS needed for decreasing step by step for factor 2
00141     int32_t tmp_width = width;
00142     int32_t tmp_height = height;
00143
00144     // -- WE START AT 1 !
00145     for (uint32_t i = 1; i < mip_levels; i++) {
00146         // WAIT for previous mip map level for being ready
00147         barrier.subresourceRange.baseMipLevel = i - 1;
00148         // HERE we TRANSITION for having a SRC format now
00149         barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
00150         barrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
00151         barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00152         barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
00153
00154         vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00155                               VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 0,
00156                               nullptr, 1, &barrier);
00157
00158         // when barrier over we can now blit :)
00159         VkImageBlit blit{};
00160
00161         // -- OFFSETS describing the 3D-dimesnion of the region
00162         blit.srcOffsets[0] = {0, 0, 0};
00163         blit.srcOffsets[1] = {tmp_width, tmp_height, 1};
00164         blit.srcSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00165         // copy from previous level
00166         blit.srcSubresource.mipLevel = i - 1;
00167         blit.srcSubresource.baseArrayLayer = 0;
00168         blit.srcSubresource.layerCount = 1;
00169         // -- OFFSETS describing the 3D-dimesnion of the region
00170         blit.dstOffsets[0] = {0, 0, 0};
00171         blit.dstOffsets[1] = {tmp_width > 1 ? tmp_width / 2 : 1,
00172                             tmp_height > 1 ? tmp_height / 2 : 1, 1};
00173         blit.dstSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00174         // -- COPY to next mipmap level
00175         blit.dstSubresource.mipLevel = i;
00176         blit.dstSubresource.baseArrayLayer = 0;
00177         blit.dstSubresource.layerCount = 1;
```

```

00178     vkCmdBlitImage(command_buffer, image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
00179                     image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &blit,
00180                     VK_FILTER_LINEAR);
00181
00182     // REARRANGE image formats for having the correct image formats again
00183     barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
00184     barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00185     barrier.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
00186     barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
00187
00188     vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00189                           VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, nullptr,
00190                           0, nullptr, 1, &barrier);
00191
00192     if (tmp_width > 1) tmp_width /= 2;
00193     if (tmp_height > 1) tmp_height /= 2;
00194 }
00195
00196     barrier.subresourceRange.baseMipLevel = mip_levels - 1;
00197     barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
00198     barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00199     barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00200     barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
00201
00202     vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
00203                           VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, nullptr, 0,
00204                           nullptr, 1, &barrier);
00205
00206     commandBufferManager.endAndSubmitCommandBuffer(device, command_pool, queue,
00207                                                   command_buffer);
00208 }
00209 }
```

## 6.199 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/scene/← Vertex.cpp File Reference

### Namespaces

- namespace [vertex](#)

### Functions

- std::array< VkVertexInputAttributeDescription, 4 > [vertex::getVertexInputAttributeDesc \(\)](#)

## 6.200 Vertex.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Vertex.hpp"
00002
00003 Vertex::Vertex() {
00004     this->pos = glm::vec3(-1.f);
00005     this->normal = glm::vec3(-1.f);
00006     this->color = glm::vec3(-1.f);
00007     this->texture_coords = glm::vec3(-1.f);
00008 }
00009
00010 Vertex::Vertex(glm::vec3 pos, glm::vec3 normal, glm::vec3 color,
00011                  glm::vec2 texture_coords) {
00012     this->pos = pos;
00013     this->normal = normal;
00014     this->color = color;
00015     this->texture_coords = texture_coords;
00016 }
00017
00018 namespace vertex {
00019
00020     std::array<VkVertexInputAttributeDescription, 4> getVertexInputAttributeDesc() {
00021         std::array<VkVertexInputAttributeDescription, 4> attribute_descriptions;
00022 }
```

```

00023 // Position attribute
00024 attribute_descriptions[0].binding = 0;
00025 attribute_descriptions[0].location = 0;
00026 attribute_descriptions[0].format =
00027     VK_FORMAT_R32G32B32_SFLOAT; // format data will take (also helps define
00028             // size of data)
00029 attribute_descriptions[0].offset = offsetof(Vertex, pos);
00030
00031 // normal coord attribute
00032 attribute_descriptions[1].binding = 0;
00033 attribute_descriptions[1].location = 1;
00034 attribute_descriptions[1].format =
00035     VK_FORMAT_R32G32B32_SFLOAT; // format data will take (also helps define
00036             // size of data)
00037 attribute_descriptions[1].offset =
00038     offsetof(Vertex, normal); // where this attribute is defined in the data
00039             // for a single vertex
00040
00041 // normal coord attribute
00042 attribute_descriptions[2].binding = 0;
00043 attribute_descriptions[2].location = 2;
00044 attribute_descriptions[2].format =
00045     VK_FORMAT_R32G32B32_SFLOAT; // format data will take (also helps define
00046             // size of data)
00047 attribute_descriptions[2].offset = offsetof(Vertex, color);
00048
00049 attribute_descriptions[3].binding = 0;
00050 // texture coord attribute
00051 attribute_descriptions[3].location = 3;
00052 attribute_descriptions[3].format =
00053     VK_FORMAT_R32G32_SFLOAT; // format data will take (also helps define size
00054             // of data)
00055 attribute_descriptions[3].offset =
00056     offsetof(Vertex, texture_coords); // where this attribute is defined in
00057             // the data for a single vertex
00058
00059 return attribute_descriptions;
00060 }
00061
00062 } // namespace vertex

```

## 6.201 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/util/← File.cpp File Reference

### 6.202 File.cpp

[Go to the documentation of this file.](#)

```

00001 #include "File.hpp"
00002
00003 #include <fstream>
00004 #include <iostream>
00005
00006 File::File(const std::string& file_location) {
00007     this->file_location = file_location;
00008 }
00009
00010 std::string File::read() {
00011     std::string content;
00012     std::ifstream file_stream(file_location, std::ios::in);
00013
00014     if (!file_stream.is_open()) {
00015         printf("Failed to read %. File does not exist.", file_location.c_str());
00016         return "";
00017     }
00018
00019     std::string line = "";
00020     while (!file_stream.eof()) {
00021         std::getline(file_stream, line);
00022         content.append(line + "\n");
00023     }
00024
00025     file_stream.close();
00026     return content;
00027 }
00028
00029 std::vector<char> File::readCharSequence() {
00030     // open stream from given file

```

```

00031 // std::ios::binary tells stream to read file as binary
00032 // std::ios::ate tells stream to start reading from end of file
00033 std::ifstream file(file_location, std::ios::binary | std::ios::ate);
00034
00035 // check if file stream sucessfully opened
00036 if (!file.is_open()) {
00037     throw std::runtime_error("Failed to open a file!");
00038 }
00039
00040 size_t file_size = (size_t)file.tellg();
00041 std::vector<char> file_buffer(file_size);
00042
00043 // move read position to start of file
00044 file.seekg(0);
00045
00046 // read the file data into the buffer (stream "file_size" in total)
00047 file.read(file_buffer.data(), file_size);
00048
00049 file.close();
00050
00051 return file_buffer;
00052 }
00053
00054 std::string File::getBaseDir() {
00055     if (file_location.find_last_of("/\\") != std::string::npos)
00056         return file_location.substr(0, file_location.find_last_of("/\\"));
00057     return "";
00058 }
00059
00060 File::~File() {}

```

## 6.203 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/ShaderHelper.cpp File Reference

### 6.204 ShaderHelper.cpp

[Go to the documentation of this file.](#)

```

00001 #include <sstream>
00002 #include <iomanip>
00003
00004 #include "ShaderHelper.hpp"
00005 #include "VulkanRendererConfig.hpp"
00006 #include "Utilities.hpp"
00007
00008 ShaderHelper::ShaderHelper() {}
00009
00010 void ShaderHelper::compileShader(const std::string& shader_src_dir,
00011                                     const std::string& shader_name) {
00012     // GLSLC_EXE is set by cmake to the location of the vulkan glslc
00013     std::stringstream shader_src_path;
00014     std::stringstream shader_log_file;
00015     std::stringstream cmdShaderCompile;
00016     std::stringstream adminPrivileges;
00017     adminPrivileges << "runas /user:<admin-user> \"";
00018
00019     // with wrapping your path with quotation marks one can use paths with blanks ...
00020     shader_src_path << shader_src_dir << shader_name;
00021     std::string shader_spv_path = getShaderSpvDir(shader_src_dir, shader_name);
00022     shader_log_file << shader_src_dir << shader_name << ".log.txt";
00023     std::stringstream log_stdout_and_stderr;
00024     log_stdout_and_stderr << " > " << shader_log_file.str() << " 2> "
00025             << shader_log_file.str();
00026
00027     cmdShaderCompile //<< adminPrivileges.str()
00028         << GLSLC_EXE << target << std::quoted(shader_src_path.str()) << " -o "
00029         << std::quoted(shader_spv_path);
00030 //<< log_stdout_and_stderr.str();
00031
00032 // std::cout << cmdShaderCompile.str().c_str();
00033
00034 system(cmdShaderCompile.str().c_str());
00035
00036 }
00037
00038 std::string ShaderHelper::getShaderSpvDir(const std::string& shader_src_dir,
00039                                              const std::string& shader_name) {
00040     std::string shader_spv_dir = "spv/";

```

```

00041
00042     std::stringstream vertShaderSpv;
00043     vertShaderSpv << shader_src_dir << shader_spv_dir << shader_name << ".spv";
00044
00045     return vertShaderSpv.str();
00046 }
00047
00048 VkShaderModule ShaderHelper::createShaderModule(VulkanDevice* device,
00049                                         const std::vector<char>& code) {
00050     // shader module create info
00051     VkShaderModuleCreateInfo shader_module_create_info{};
00052     shader_module_create_info.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
00053     shader_module_create_info.codeSize = code.size(); // size of code
00054     shader_module_create_info.pCode =
00055         reinterpret_cast<const uint32_t*>(code.data()); // pointer to code
00056
00057     VkShaderModule shader_module;
00058     VkResult result =
00059         vkCreateShaderModule(device->getLogicalDevice(),
00060                               &shader_module_create_info, nullptr, &shader_module);
00061
00062     ASSERT_VULKAN(result, "Failed to create a shader module!")
00063
00064     return shader_module;
00065 }
00066
00067 ShaderHelper::~ShaderHelper() {}

```

## 6.205 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanBuffer.cpp File Reference

### 6.206 VulkanBuffer.cpp

[Go to the documentation of this file.](#)

```

00001 #include "VulkanBuffer.hpp"
00002
00003 #include <stdexcept>
00004
00005 #include "MemoryHelper.hpp"
00006 #include "Utilities.hpp"
00007
00008 VulkanBuffer::VulkanBuffer() {}
00009
00010 void VulkanBuffer::create(VulkanDevice* device, VkDeviceSize buffer_size,
00011                             VkBufferUsageFlags buffer_usage_flags,
00012                             VkMemoryPropertyFlags buffer_property_flags) {
00013     this->device = device;
00014
00015     // information to create a buffer (doesn't include assigning memory)
00016     VkBufferCreateInfo buffer_info{};
00017     buffer_info.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
00018     buffer_info.size = buffer_size;
00019     // multiple types of buffer possible, e.g. vertex buffer
00020     buffer_info.usage = buffer_usage_flags;
00021     // similar to swap chain images, can share vertex buffers
00022     buffer_info.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
00023
00024     VkResult result = vkCreateBuffer(device->getLogicalDevice(), &buffer_info,
00025                                     nullptr, &buffer);
00026     ASSERT_VULKAN(result, "Failed to create a buffer!");
00027
00028     // get buffer memory requirements
00029     VkMemoryRequirements memory_requirements{};
00030     vkGetBufferMemoryRequirements(device->getLogicalDevice(), buffer,
00031                                   &memory_requirements);
00032
00033     // allocate memory to buffer
00034     VkMemoryAllocateInfo memory_alloc_info{};
00035     memory_alloc_info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
00036     memory_alloc_info.allocationSize = memory_requirements.size;
00037
00038     uint32_t memory_type_index = find_memory_type_index(
00039         device->getPhysicalDevice(), memory_requirements.memoryTypeBits,
00040         buffer_property_flags);
00041
00042     // VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | /* memory is visible to
00043     // CPU side

```

```

00044 // */ VK_MEMORY_PROPERTY_HOST_COHERENT_BIT /* data is placed straight into
00045 // buffer */;
00046 if (memory_type_index < 0) {
00047     throw std::runtime_error("Failed to find suitable memory type!");
00048 }
00049 memory_alloc_info.memoryTypeIndex = memory_type_index;
00050
00052 // allocate memory to VkDeviceMemory
00053 result = vkAllocateMemory(device->getLogicalDevice(), &memory_alloc_info,
00054                         nullptr, &bufferMemory);
00055 ASSERT_VULKAN(result, "Failed to allocate memory for buffer!");
00056
00057 // allocate memory to given buffer
00058 vkBindBufferMemory(device->getLogicalDevice(), buffer, bufferMemory, 0);
00059
00060 created = true;
00061 }
00062
00063 void VulkanBuffer::cleanUp() {
00064     if (created) {
00065         vkDestroyBuffer(device->getLogicalDevice(), buffer, nullptr);
00066         vkFreeMemory(device->getLogicalDevice(), bufferMemory, nullptr);
00067     }
00068 }
00069
00070 VulkanBuffer::~VulkanBuffer() {}

```

## 6.207 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanBufferManager.cpp File Reference

### 6.208 VulkanBufferManager.cpp

[Go to the documentation of this file.](#)

```

00001 #include "VulkanBufferManager.hpp"
00002
00003 VulkanBufferManager::VulkanBufferManager() {}
00004
00005 void VulkanBufferManager::copyBuffer(VkDevice device, VkQueue transfer_queue,
00006                                         VkCommandPool transfer_command_pool,
00007                                         VulkanBuffer src_buffer,
00008                                         VulkanBuffer dst_buffer,
00009                                         VkDeviceSize buffer_size) {
0010     // create buffer
0011     VkCommandBuffer command_buffer =
0012         commandBufferManager.beginCommandBuffer(device, transfer_command_pool);
0013
0014     // region of data to copy from and to
0015     VkBufferCopy buffer_copy_region{};
0016     buffer_copy_region.srcOffset = 0;
0017     buffer_copy_region.dstOffset = 0;
0018     buffer_copy_region.size = buffer_size;
0019
0020     // command to copy src buffer to dst buffer
0021     vkCmdCopyBuffer(command_buffer, src_buffer.getBuffer(),
0022                      dst_buffer.getBuffer(), 1, &buffer_copy_region);
0023
0024     commandBufferManager.endAndSubmitCommandBuffer(
0025         device, transfer_command_pool, transfer_queue, command_buffer);
0026 }
0027
0028 void VulkanBufferManager::copyImageBuffer(VkDevice device,
0029                                             VkQueue transfer_queue,
0030                                             VkCommandPool transfer_command_pool,
0031                                             VulkanBuffer src_buffer, VkImage image,
0032                                             uint32_t width, uint32_t height) {
0033     // create buffer
0034     VkCommandBuffer transfer_command_buffer =
0035         commandBufferManager.beginCommandBuffer(device, transfer_command_pool);
0036
0037     VkBufferImageCopy image_region{};
0038     image_region.bufferOffset = 0; // offset into data
0039     image_region.bufferRowLength =
0040         0; // row length of data to calculate data spacing
0041     image_region.bufferImageHeight = 0; // image height to calculate data spacing
0042     image_region.imageSubresource.aspectMask =
0043         VK_IMAGE_ASPECT_COLOR_BIT; // which aspect of image to copy

```

```

00044     image_region.imageSubresource.mipLevel = 0;
00045     image_region.imageSubresource.baseArrayLayer = 0;
00046     image_region.imageSubresource.layerCount = 1;
00047     image_region.imageOffset = {0, 0, 0}; // offset into image
00048     image_region.imageExtent = {width, height, 1};
00049
00050     // copy buffer to given image
00051     vkCmdCopyBufferToImage(transfer_command_buffer, src_buffer, image,
00052                             VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1,
00053                             &image_region);
00054
00055     commandBufferManager.endAndSubmitCommandBuffer(
00056         device, transfer_command_pool, transfer_queue, transfer_command_buffer);
00057 }
00058
00059 VulkanBufferManager::~VulkanBufferManager() {}

```

## 6.209 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanDebug.cpp File Reference

### Namespaces

- namespace [debug](#)

### Functions

- VKAPI\_ATTR VkBool32 VKAPI\_CALL [debug::debugUtilsMessengerCallback](#) (VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity, VkDebugUtilsMessageTypeFlagsEXT messageType, const VkDebugUtilsMessengerCallbackDataEXT \*pCallbackData, void \*pUserData)
- void [debug::setupDebugging](#) (VkInstance instance, VkDebugReportFlagsEXT flags, VkDebugReportCallbackEXT callBack)
- void [debug::freeDebugCallback](#) (VkInstance instance)

### Variables

- PFN\_vkCreateDebugUtilsMessengerEXT [debug::vkCreateDebugUtilsMessengerEXT](#)
- PFN\_vkDestroyDebugUtilsMessengerEXT [debug::vkDestroyDebugUtilsMessengerEXT](#)
- VkDebugUtilsMessengerEXT [debug::debugUtilsMessenger](#)

## 6.210 VulkanDebug.cpp

[Go to the documentation of this file.](#)

```

00001 #include "VulkanDebug.hpp"
00002
00003 #include "Utilities.hpp"
00004
00005 namespace debug {
00006 PFN_vkCreateDebugUtilsMessengerEXT vkCreateDebugUtilsMessengerEXT;
00007 PFN_vkDestroyDebugUtilsMessengerEXT vkDestroyDebugUtilsMessengerEXT;
00008 VkDebugUtilsMessengerEXT debugUtilsMessenger;
00009
0010 VKAPI_ATTR VkBool32 VKAPI_CALL debugUtilsMessengerCallback(
0011     VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
0012     VkDebugUtilsMessageTypeFlagsEXT messageType,
0013     const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
0014     void* pUserData) {
0015     // Select prefix depending on flags passed to the callback
0016     std::string prefix("");
0017
0018     if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT) {
0019         prefix = "VERBOSE: ";

```

```

00020 } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT) {
00021     prefix = "INFO: ";
00022 } else if (messageSeverity &
00023             VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
00024     prefix = "WARNING: ";
00025 } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
00026     prefix = "ERROR: ";
00027 }
00028
00029 // Display message to default output (console/logcat)
00030 std::stringstream debugMessage;
00031 debugMessage << prefix << "[" << pCallbackData->messageIdNumber << "]["
00032             << pCallbackData->pMessageIdName
00033             << "] : " << pCallbackData->pMessage;
00034
00035 #if defined(__ANDROID__)
00036     if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
00037         LOGE("%s", debugMessage.str().c_str());
00038     } else {
00039         LOGD("%s", debugMessage.str().c_str());
00040     }
00041 #else
00042     if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
00043         std::cerr << debugMessage.str() << "\n";
00044     } else {
00045         std::cout << debugMessage.str() << "\n";
00046     }
00047     fflush(stdout);
00048 #endif
00049
00050 // The return value of this callback controls whether the Vulkan call that
00051 // caused the validation message will be aborted or not. We return VK_FALSE as
00052 // we DON'T want Vulkan calls that cause a validation message to abort. If you
00053 // instead want to have calls abort, pass in VK_TRUE and the function will
00054 // return VK_ERROR_VALIDATION_FAILED_EXT
00055     return VK_FALSE;
00056 }
00057
00058 void setupDebugging(VkInstance instance, VkDebugReportFlagsEXT flags,
00059                      VkDebugReportCallbackEXT callBack) {
00060     vkCreateDebugUtilsMessengerEXT =
00061         reinterpret_cast<PFN_vkCreateDebugUtilsMessengerEXT>(
00062             vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT"));
00063     vkDestroyDebugUtilsMessengerEXT =
00064         reinterpret_cast<PFN_vkDestroyDebugUtilsMessengerEXT>(
00065             vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT"));
00066
00067     VkDebugUtilsMessengerCreateInfoEXT debugUtilsMessengerCI{};
00068     debugUtilsMessengerCI.sType =
00069         VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
00070     debugUtilsMessengerCI.messageSeverity =
00071         VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
00072         VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
00073     debugUtilsMessengerCI.messageType =
00074         VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
00075         VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT;
00076     debugUtilsMessengerCI.pfnUserCallback = debugUtilsMessengerCallback;
00077     ASSERT_VULKAN(vkCreateDebugUtilsMessengerEXT(instance, &debugUtilsMessengerCI,
00078                                                 nullptr, &debugUtilsMessenger),
00079                   "Failed to create debug messenger")
00080 }
00081
00082 void freeDebugCallback(VkInstance instance) {
00083     if (debugUtilsMessenger != VK_NULL_HANDLE) {
00084         vkDestroyDebugUtilsMessengerEXT(instance, debugUtilsMessenger, nullptr);
00085     }
00086 }
00087 } // namespace debug

```

## 6.211 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanDevice.cpp File Reference

## 6.212 VulkanDevice.cpp

[Go to the documentation of this file.](#)

```

00001 #include "VulkanDevice.hpp"
00002

```

```
00003 #include <string.h>
00004
00005 #include <set>
00006 #include <string>
00007
00008 VulkanDevice::VulkanDevice(VulkanInstance* instance, VkSurfaceKHR* surface) {
00009     this->instance = instance;
00010     this->surface = surface;
00011     get_physical_device();
00012     create_logical_device();
00013 }
00014
00015 SwapChainDetails VulkanDevice::getSwapchainDetails() {
00016     return getSwapchainDetails(physical_device);
00017 }
00018
00019 void VulkanDevice::cleanUp() { vkDestroyDevice(logical_device, nullptr); }
00020
00021 VulkanDevice::~VulkanDevice() {}
00022
00023 QueueFamilyIndices VulkanDevice::getQueueFamilies() {
00024     QueueFamilyIndices indices{};
00025
00026     uint32_t queue_family_count = 0;
00027     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
00028                                             nullptr);
00029
00030     std::vector<VkQueueFamilyProperties> queue_family_list(queue_family_count);
00031     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
00032                                             queue_family_list.data());
00033
00034     // Go through each queue family and check if it has at least 1 of required
00035     // types we need to keep track th eindex by our own
00036     int index = 0;
00037     for (const auto& queue_family : queue_family_list) {
00038         // first check if queue family has at least 1 queue in that family
00039         // Queue can be multiple types defined through bitfield. Need to bitwise AND
00040         // with VK_QUE_*_BIT to check if has required type
00041         if (queue_family.queueCount > 0 &&
00042             queue_family.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
00043             indices.graphics_family = index; // if queue family valid, than get index
00044         }
00045
00046         if (queue_family.queueCount > 0 &&
00047             queue_family.queueFlags & VK_QUEUE_COMPUTE_BIT) {
00048             indices.compute_family = index;
00049         }
00050
00051         // check if queue family supports presentation
00052         VkBool32 presentation_support = false;
00053         vkGetPhysicalDeviceSurfaceSupportKHR(physical_device, index, *surface,
00054                                             &presentation_support);
00055         // check if queue is presentation type (can be both graphics and
00056         // presentation)
00057         if (queue_family.queueCount > 0 && presentation_support) {
00058             indices.presentation_family = index;
00059         }
00060
00061         // check if queue family indices are in a valid state
00062         if (indices.is_valid()) {
00063             break;
00064         }
00065
00066         index++;
00067     }
00068
00069     return indices;
00070 }
00071
00072 void VulkanDevice::get_physical_device() {
00073     // Enumerate physical devices the vkInstance can access
00074     uint32_t device_count = 0;
00075     vkEnumeratePhysicalDevices(instance->getVulkanInstance(), &device_count,
00076                               nullptr);
00077
00078     // if no devices available, then none support of Vulkan
00079     if (device_count == 0) {
00080         throw std::runtime_error(
00081             "Can not find GPU's that support Vulkan Instance!");
00082     }
00083
00084     // Get list of physical devices
00085     std::vector<VkPhysicalDevice> device_list(device_count);
00086     vkEnumeratePhysicalDevices(instance->getVulkanInstance(), &device_count,
00087                               device_list.data());
00088
00089     for (const auto& device : device_list) {
```

```

00090     if (check_device_suitable(device)) {
00091         physical_device = device;
00092         break;
00093     }
00094 }
00095
00096 // get properties of our new device
00097 vkGetPhysicalDeviceProperties(physical_device, &device_properties);
00098 }
00099
00100 void VulkanDevice::create_logical_device() {
00101     // get the queue family indices for the chosen physical device
00102     QueueFamilyIndices indices = getQueueFamilies();
00103
00104     // vector for queue creation information and set for family indices
00105     std::vector<VkDeviceQueueCreateInfo> queue_create_infos;
00106     std::set<int> queue_family_indices = {indices.graphics_family,
00107                                         indices.presentation_family,
00108                                         indices.compute_family};
00109
00110     // Queue the logical device needs to create and info to do so (only 1 for now,
00111     // will add more later!)
00112     for (int queue_family_index : queue_family_indices) {
00113         VkDeviceQueueCreateInfo queue_create_info{};
00114         queue_create_info.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
00115         queue_create_info.queueFamilyIndex =
00116             queue_family_index; // the index of the family to create a queue from
00117         queue_create_info.queueCount = 1; // number of queues to create
00118         float priority = 1.0f;
00119         queue_create_info.pQueuePriorities =
00120             &priority; // Vulkan needs to know how to handle multiple queues, so
00121             // decide priority (1 = highest)
00122
00123         queue_create_infos.push_back(queue_create_info);
00124     }
00125
00126     // -- ALL EXTENSION WE NEED
00127     VkPhysicalDeviceDescriptorIndexingFeatures indexing_features{};
00128     indexing_features.sType =
00129         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES;
00130     indexing_features.runtimeDescriptorArray = VK_TRUE;
00131     indexing_features.shaderSampledImageArrayNonUniformIndexing = VK_TRUE;
00132     indexing_features.pNext = nullptr;
00133
00134     // -- NEEDED FOR QUERING THE DEVICE ADDRESS WHEN CREATING ACCELERATION
00135     // STRUCTURES
00136     VkPhysicalDeviceBufferDeviceAddressFeaturesEXT
00137         buffer_device_address_features{};
00138     buffer_device_address_features.sType =
00139         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT;
00140     buffer_device_address_features.pNext = &indexing_features;
00141     buffer_device_address_features.bufferDeviceAddress = VK_TRUE;
00142     buffer_device_address_features.bufferDeviceAddressCaptureReplay = VK_TRUE;
00143     buffer_device_address_features.bufferDeviceAddressMultiDevice = VK_FALSE;
00144
00145     // --ENABLE RAY TRACING PIPELINE
00146     VkPhysicalDeviceRayTracingPipelineFeaturesKHR ray_tracing_pipeline_features{};
00147     ray_tracing_pipeline_features.sType =
00148         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_FEATURES_KHR;
00149     ray_tracing_pipeline_features.pNext = &buffer_device_address_features;
00150     ray_tracing_pipeline_features.rayTracingPipeline = VK_TRUE;
00151
00152     // -- ENABLE ACCELERATION STRUCTURES
00153     VkPhysicalDeviceAccelerationStructureFeaturesKHR
00154         acceleration_structure_features{};
00155     acceleration_structure_features.sType =
00156         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_FEATURES_KHR;
00157     acceleration_structure_features.pNext = &ray_tracing_pipeline_features;
00158     acceleration_structure_features.accelerationStructure = VK_TRUE;
00159     acceleration_structure_features.accelerationStructureCaptureReplay = VK_TRUE;
00160     acceleration_structure_features.accelerationStructureIndirectBuild = VK_FALSE;
00161     acceleration_structure_features.accelerationStructureHostCommands = VK_FALSE;
00162     acceleration_structure_features
00163         .descriptorBindingAccelerationStructureUpdateAfterBind = VK_FALSE;
00164
00165     VkPhysicalDeviceVulkan13Features features13{};
00166     features13.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES;
00167     features13.maintenance4 = VK_TRUE;
00168     features13.robustImageAccess = VK_FALSE;
00169     features13.inlineUniformBlock = VK_FALSE;
00170     features13.descriptorBindingInlineUniformBlockUpdateAfterBind = VK_FALSE;
00171     features13.pipelineCreationCacheControl = VK_FALSE;
00172     features13.privateData = VK_FALSE;
00173     features13.shaderDemoteToHelperInvocation = VK_FALSE;
00174     features13.shaderTerminateInvocation = VK_FALSE;
00175     features13.subgroupSizeControl = VK_FALSE;
00176     features13.computeFullSubgroups = VK_FALSE;

```

```

00177     features13.synchronization2 = VK_FALSE;
00178     features13.textureCompressionASTC_HDR = VK_FALSE;
00179     features13.shaderZeroInitializeWorkgroupMemory = VK_FALSE;
00180     features13.dynamicRendering = VK_FALSE;
00181     features13.shaderIntegerDotProduct = VK_FALSE;
00182     features13.pNext = &acceleration_structure_features;
00183
00184     VkPhysicalDeviceRayQueryFeaturesKHR rayQueryFeature{};
00185     rayQueryFeature.sType =
00186         VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_QUERY_FEATURES_KHR;
00187     rayQueryFeature.pNext = &features13;
00188     rayQueryFeature.rayQuery = VK_TRUE;
00189
00190     VkPhysicalDeviceFeatures2 features2{};
00191     features2.pNext = &rayQueryFeature;
00192     features2.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2;
00193     features2.features.samplerAnisotropy = VK_TRUE;
00194     features2.features.shaderInt64 = VK_TRUE;
00195     features2.features.geometryShader = VK_TRUE;
00196     features2.features.logicOp = VK_TRUE;
00197
00198 // -- PREPARE FOR HAVING MORE EXTENSION BECAUSE WE NEED RAYTRACING
00199 // CAPABILITIES
00200 std::vector<const char*> extensions(device_extensions);
00201
00202 // COPY ALL NECESSARY EXTENSIONS FOR RAYTRACING TO THE EXTENSION
00203 extensions.insert(extensions.begin(),
00204     device_extensions_for_raytracing.begin(),
00205     device_extensions_for_raytracing.end());
00206
00207 // information to create logical device (sometimes called "device")
00208 VkDeviceCreateInfo device_create_info{};
00209 device_create_info.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
00210 device_create_info.queueCreateInfoCount = static_cast<uint32_t>(
00211     queue_create_infos.size()); // number of queue create infos
00212 device_create_info.pQueueCreateInfos =
00213     queue_create_info.data(); // list of queue create infos so device can
00214         // create required queues
00215 device_create_info.enabledExtensionCount = static_cast<uint32_t>(
00216     extensions.size()); // number of enabled logical device extensions
00217 device_create_info.ppEnabledExtensionNames =
00218     extensions.data(); // list of enabled logical device extensions
00219 device_create_info.flags = 0;
00220 device_create_info.pEnabledFeatures = NULL;
00221
00222 device_create_info.pNext = &features2;
00223
00224 // create logical device for the given physical device
00225 VkResult result = vkCreateDevice(physical_device, &device_create_info,
00226         nullptr, &logical_device);
00227 ASSERT_VULKAN(result, "Failed to create a logical device!");
00228
00229 // Queues are created at the same time as the device...
00230 // So we want handle to queues
00231 // From given logical device of given queue family, of given queue index (0
00232 // since only one queue), place reference in given VkQueue
00233 vkGetDeviceQueue(logical_device, indices.graphics_family, 0, &graphics_queue);
00234 vkGetDeviceQueue(logical_device, indices.presentation_family, 0,
00235         &presentation_queue);
00236 vkGetDeviceQueue(logical_device, indices.compute_family, 0, &compute_queue);
00237 }
00238
00239 QueueFamilyIndices VulkanDevice::getQueueFamilies(
00240     VkPhysicalDevice physical_device) {
00241     QueueFamilyIndices indices();
00242
00243     uint32_t queue_family_count = 0;
00244     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
00245         nullptr);
00246
00247     std::vector<VkQueueFamilyProperties> queue_family_list(queue_family_count);
00248     vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
00249         queue_family_list.data());
00250
00251 // Go through each queue family and check if it has at least 1 of required
00252 // types we need to keep track th eindex by our own
00253     int index = 0;
00254     for (const auto& queue_family : queue_family_list) {
00255         // first check if queue family has at least 1 queue in that family
00256         // Queue can be multiple types defined through bitfield. Need to bitwise AND
00257         // with VK_QUE_*_BIT to check if has required type
00258         if (queue_family.queueCount > 0 &&
00259             queue_family.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
00260             indices.graphics_family = index; // if queue family valid, than get index
00261         }
00262
00263         if (queue_family.queueCount > 0 &&

```

```

00264     queue_family.queueFlags & VK_QUEUE_COMPUTE_BIT) {
00265         indices.compute_family = index;
00266     }
00267
00268     // check if queue family supports presentation
00269     VkBool32 presentation_support = false;
00270     vkGetPhysicalDeviceSurfaceSupportKHR(physical_device, index, *surface,
00271                                         &presentation_support);
00272     // check if queue is presentation type (can be both graphics and
00273     // presentation)
00274     if (queue_family.queueCount > 0 && presentation_support) {
00275         indices.presentation_family = index;
00276     }
00277
00278     // check if queue family indices are in a valid state
00279     if (indices.is_valid()) {
00280         break;
00281     }
00282
00283     index++;
00284 }
00285
00286 return indices;
00287 }
00288
00289 SwapChainDetails VulkanDevice::getSwapchainDetails(VkPhysicalDevice device) {
00290     SwapChainDetails swapchain_details{};
00291     // get the surface capabilities for the given surface on the given physical
00292     // device
00293     vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
00294         device, *surface, &swapchain_details.surface_capabilities);
00295
00296     uint32_t format_count = 0;
00297     vkGetPhysicalDeviceSurfaceFormatsKHR(device, *surface, &format_count,
00298                                         nullptr);
00299
00300     // if formats returned, get list of formats
00301     if (format_count != 0) {
00302         swapchain_details.formats.resize(format_count);
00303         vkGetPhysicalDeviceSurfaceFormatsKHR(device, *surface, &format_count,
00304                                             swapchain_details.formats.data());
00305     }
00306
00307     uint32_t presentation_count = 0;
00308     vkGetPhysicalDeviceSurfacePresentModesKHR(device, *surface,
00309                                              &presentation_count, nullptr);
00310
00311     // if presentation modes returned, get list of presentation modes
00312     if (presentation_count > 0) {
00313         swapchain_details.presentation_mode.resize(presentation_count);
00314         vkGetPhysicalDeviceSurfacePresentModesKHR(
00315             device, *surface, &presentation_count,
00316             swapchain_details.presentation_mode.data());
00317     }
00318
00319     return swapchain_details;
00320 }
00321
00322 bool VulkanDevice::check_device_suitable(VkPhysicalDevice device) {
00323     // Information about device itself (ID, name, type, vendor, etc)
00324     VkPhysicalDeviceProperties device_properties;
00325     vkGetPhysicalDeviceProperties(device, &device_properties);
00326
00327     VkPhysicalDeviceFeatures device_features;
00328     vkGetPhysicalDeviceFeatures(device, &device_features);
00329
00330     QueueFamilyIndices indices = getQueueFamilies(device);
00331
00332     bool extensions_supported = check_device_extension_support(device);
00333
00334     bool swap_chain_valid = false;
00335
00336     if (extensions_supported) {
00337         SwapChainDetails swap_chain_details = getSwapchainDetails(device);
00338         swap_chain_valid = !swap_chain_details.presentation_mode.empty() &&
00339                            !swap_chain_details.formats.empty();
00340     }
00341
00342     return indices.is_valid() && extensions_supported && swap_chain_valid &&
00343             device_features.samplerAnisotropy;
00344 }
00345
00346 bool VulkanDevice::check_device_extension_support(VkPhysicalDevice device) {
00347     uint32_t extension_count = 0;
00348     vkEnumerateDeviceExtensionProperties(device, nullptr, &extension_count,
00349                                         nullptr);
00350

```

```

00351     if (extension_count == 0) {
00352         return false;
00353     }
00354
00355     // populate list of extensions
00356     std::vector<VkExtensionProperties> extensions(extension_count);
00357     vkEnumerateDeviceExtensionProperties(device, nullptr, &extension_count,
00358                                         extensions.data());
00359
00360     for (const auto& device_extension : device_extensions) {
00361         bool has_extension = false;
00362
00363         for (const auto& extension : extensions) {
00364             if (strcmp(device_extension, extension.extensionName) == 0) {
00365                 has_extension = true;
00366                 break;
00367             }
00368         }
00369
00370         if (!has_extension) {
00371             return false;
00372         }
00373     }
00374
00375     return true;
00376 }
```

## 6.213 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanImage.cpp File Reference

### 6.214 VulkanImage.cpp

[Go to the documentation of this file.](#)

```

00001 #include "VulkanImage.hpp"
00002
00003 #include "MemoryHelper.hpp"
00004 #include "Utilities.hpp"
00005
00006 VulkanImage::VulkanImage() {}
00007
00008 void VulkanImage::create(VulkanDevice* device, uint32_t width, uint32_t height,
00009                             uint32_t mip_levels, VkFormat format,
00010                             VkImageTiling tiling, VkImageUsageFlags use_flags,
00011                             VkMemoryPropertyFlags prop_flags) {
00012     this->device = device;
00013     // CREATE image
00014     // image creation info
00015     VkImageCreateInfo image_create_info{};
00016     image_create_info.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
00017     image_create_info.imageType = VK_IMAGE_TYPE_2D; // type of image (1D, 2D, 3D)
00018     image_create_info.extent.width = width; // width if image extent
00019     image_create_info.extent.height = height; // height if image extent
00020     image_create_info.extent.depth = 1; // height if image extent
00021     image_create_info.mipLevels = mip_levels; // number of mipmap levels
00022     image_create_info.arrayLayers = 1; // number of levels in image array
00023     image_create_info.format = format; // format type of image
00024     image_create_info.tiling =
00025         tiling; // tiling of image ("arranged" for optimal reading)
00026     image_create_info.initialLayout =
00027         VK_IMAGE_LAYOUT_UNDEFINED; // layout of image data on creation
00028     image_create_info.usage =
00029         use_flags; // bit flags defining what image will be used for
00030     image_create_info.samples =
00031         VK_SAMPLE_COUNT_1_BIT; // number of samples for multisampling
00032     image_create_info.sharingMode =
00033         VK_SHARING_MODE_EXCLUSIVE; // whether image can be shared between queues
00034
00035     VkResult result = vkCreateImage(device->getLogicalDevice(),
00036                                     &image_create_info, nullptr, &image);
00037     ASSERT_VULKAN(result, "Failed to create an image!")
00038
00039     // CREATE memory for image
00040     // get memory requirements for a type of image
00041     VkMemoryRequirements memory_requirements;
00042     vkGetImageMemoryRequirements(device->getLogicalDevice(), image,
00043                                  &memory_requirements);
00044 }
```

```

00045 // allocate memory using image requirements and user defined properties
00046 VkMemoryAllocateInfo memory_alloc_info{};
00047 memory_alloc_info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
00048 memory_alloc_info.allocationSize = memory_requirements.size;
00049 memory_alloc_info.memoryTypeIndex =
00050     find_memory_type_index(device->getPhysicalDevice(),
00051                             memory_requirements.memoryTypeBits, prop_flags);
00052
00053 result = vkAllocateMemory(device->getLogicalDevice(), &memory_alloc_info,
00054                             nullptr, &imageMemory);
00055 ASSERT_VULKAN(result, "Failed to allocate memory!")
00056
00057 // connect memory to image
00058 vkBindImageMemory(device->getLogicalDevice(), image, imageMemory, 0);
00059 }
00060
00061 void VulkanImage::transitionImageLayout(VkDevice device, VkQueue queue,
00062                                         VkCommandPool command_pool,
00063                                         VkImageLayout old_layout,
00064                                         VkImageLayout new_layout,
00065                                         VkImageAspectFlags aspectMask,
00066                                         uint32_t mip_levels) {
00067     VkCommandBuffer command_buffer =
00068         commandBufferManager.beginCommandBuffer(device, command_pool);
00069
00070 // VK_IMAGE_ASPECT_COLOR_BIT
00071     VkImageMemoryBarrier memory_barrier{};
00072     memory_barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00073     memory_barrier.oldLayout = old_layout;
00074     memory_barrier.newLayout = new_layout;
00075     memory_barrier.srcQueueFamilyIndex =
00076         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition from
00077     memory_barrier.dstQueueFamilyIndex =
00078         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition to
00079     memory_barrier.image =
00080         image; // image being accessed and modified as part of barrier
00081     memory_barrier.subresourceRange.aspectMask =
00082         aspectMask; // aspect of image being altered
00083     memory_barrier.subresourceRange.baseMipLevel =
00084         0; // first mip level to start alterations on
00085     memory_barrier.subresourceRange.levelCount =
00086         mip_levels; // number of mip levels to alter starting from baseMipLevel
00087     memory_barrier.subresourceRange.baseArrayLayer =
00088         0; // first layer to start alterations on
00089     memory_barrier.subresourceRange.layerCount =
00090         1; // number of layers to alter starting from baseArrayLayer
00091
00092 // if transitioning from new image to image ready to receive data
00093     memory_barrier.srcAccessMask = accessFlagsForImageLayout(old_layout);
00094     memory_barrier.dstAccessMask = accessFlagsForImageLayout(new_layout);
00095
00096     VkPipelineStageFlags src_stage = pipelineStageForLayout(old_layout);
00097     VkPipelineStageFlags dst_stage = pipelineStageForLayout(new_layout);
00098
00099     vkCmdPipelineBarrier(
00100
00101         command_buffer, src_stage,
00102         dst_stage, // pipeline stages (match to src and dst accessmask)
00103         0, // no dependency flags
00104         0,
00105         nullptr, // memory barrier count + data
00106         0,
00107         nullptr, // buffer memory barrier count + data
00108         1,
00109         &memory_barrier // image memory barrier count + data
00110
00111     );
00112
00113     commandBufferManager.endAndSubmitCommandBuffer(device, command_pool, queue,
00114                                                 command_buffer);
00115 }
00116
00117 void VulkanImage::transitionImageLayout(VkCommandBuffer command_buffer,
00118                                         VkImageLayout old_layout,
00119                                         VkImageLayout new_layout,
00120                                         uint32_t mip_levels,
00121                                         VkImageAspectFlags aspectMask) {
00122     VkImageMemoryBarrier memory_barrier{};
00123     memory_barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00124     memory_barrier.oldLayout = old_layout;
00125     memory_barrier.newLayout = new_layout;
00126     memory_barrier.srcQueueFamilyIndex =
00127         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition from
00128     memory_barrier.dstQueueFamilyIndex =
00129         VK_QUEUE_FAMILY_IGNORED; // Queue family to transition to
00130     memory_barrier.image =
00131         image; // image being accessed and modified as part of barrier

```

```

00132     memory_barrier.subresourceRange.aspectMask =
00133         aspectMask; // aspect of image being altered
00134     memory_barrier.subresourceRange.baseMipLevel =
00135         0; // first mip level to start alterations on
00136     memory_barrier.subresourceRange.levelCount =
00137         mip_levels; // number of mip levels to alter starting from baseMipLevel
00138     memory_barrier.subresourceRange.baseArrayLayer =
00139         0; // first layer to start alterations on
00140     memory_barrier.subresourceRange.layerCount =
00141         1; // number of layers to alter starting from baseArrayLayer
00142
00143     memory_barrier.srcAccessMask = accessFlagsForImageLayout(old_layout);
00144     memory_barrier.dstAccessMask = accessFlagsForImageLayout(new_layout);
00145
00146     VkPipelineStageFlags src_stage = pipelineStageForLayout(old_layout);
00147     VkPipelineStageFlags dst_stage = pipelineStageForLayout(new_layout);
00148
00149     // if transitioning from new image to image ready to receive data
00150
00151     vkCmdPipelineBarrier(
00152
00153         command_buffer, src_stage,
00154         dst_stage, // pipeline stages (match to src and dst accessmask)
00155         0,           // no dependency flags
00156         0,
00157         nullptr,    // memory barrier count + data
00158         0,
00159         nullptr,    // buffer memory barrier count + data
00160         1,
00161         &memory_barrier // image memory barrier count + data
00162
00163     );
00164 }
00165
00166 void VulkanImage::setImage(VkImage image) { this->image = image; }
00167
00168 void VulkanImage::cleanUp() {
00169     vkDestroyImage(device->getLogicalDevice(), image, nullptr);
00170     vkFreeMemory(device->getLogicalDevice(), imageMemory, nullptr);
00171 }
00172
00173 VulkanImage::~VulkanImage() {}
00174
00175 VkAccessFlags VulkanImage::accessFlagsForImageLayout(VkImageLayout layout) {
00176     switch (layout) {
00177         case VK_IMAGE_LAYOUT_PREINITIALIZED:
00178             return VK_ACCESS_HOST_WRITE_BIT;
00179         case VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:
00180             return VK_ACCESS_TRANSFER_WRITE_BIT;
00181         case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
00182             return VK_ACCESS_TRANSFER_READ_BIT;
00183         case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
00184             return VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
00185         case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
00186             return VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
00187         case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
00188             return VK_ACCESS_SHADER_READ_BIT;
00189         default:
00190             return VkAccessFlags();
00191     }
00192 }
00193
00194 VkPipelineStageFlags VulkanImage::pipelineStageForLayout(
00195     VkImageLayout oldImageLayout) {
00196     switch (oldImageLayout) {
00197         case VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:
00198         case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
00199             return VK_PIPELINE_STAGE_TRANSFER_BIT;
00200         case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
00201             return VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00202         case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
00203             return VK_PIPELINE_STAGE_ALL_COMMANDS_BIT; // We do this to allow queue
00204                                         // other than graphic return
00205                                         // VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
00206         case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
00207             return VK_PIPELINE_STAGE_ALL_COMMANDS_BIT; // We do this to allow queue
00208                                         // other than graphic return
00209                                         // VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
00210         case VK_IMAGE_LAYOUT_PREINITIALIZED:
00211             return VK_PIPELINE_STAGE_HOST_BIT;
00212         case VK_IMAGE_LAYOUT_UNDEFINED:
00213             return VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
00214         default:
00215             return VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
00216     }
00217 }

```

## 6.215 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanImageView.cpp File Reference

### 6.216 VulkanImageView.cpp

[Go to the documentation of this file.](#)

```
00001 #include "VulkanImageView.hpp"
00002
00003 VulkanImageView::VulkanImageView() {}
00004
00005 void VulkanImageView::setImageView(VkImageView imageView) {
00006     this->imageView = imageView;
00007 }
00008
00009 void VulkanImageView::create(VulkanDevice* device, VkImage image,
00010                                 VkFormat format, VkImageAspectFlags aspect_flags,
00011                                 uint32_t mip_levels) {
00012     this->device = device;
00013
00014     VkImageViewCreateInfo view_create_info{};
00015     view_create_info.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
00016     view_create_info.image = image; // image to create view for
00017     view_create_info.viewType = VK_IMAGE_VIEW_TYPE_2D; // typ of image
00018     view_create_info.format = format;
00019     view_create_info.components.r =
00020         VK_COMPONENT_SWIZZLE_IDENTITY; // allows remapping of rgba components to
00021                                     // other rgba values
00022     view_create_info.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
00023     view_create_info.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
00024     view_create_info.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
00025
00026     // subresources allow the view to view only a part of an image
00027     view_create_info.subresourceRange.aspectMask =
00028         aspect_flags; // which aspect of an image to view (e.g. color bit for
00029                     // viewing color)
00030     view_create_info.subresourceRange.baseMipLevel =
00031         0; // start mipmap level to view from
00032     view_create_info.subresourceRange.levelCount =
00033         mip_levels; // number of mipmap levels to view
00034     view_create_info.subresourceRange.baseArrayLayer =
00035         0; // start array level to view from
00036     view_create_info.subresourceRange.layerCount =
00037         1; // number of array levels to view
00038
00039     // create image view
00040     VkResult result = vkCreateImageView(device->getLogicalDevice(),
00041                                         &view_create_info, nullptr, &imageView);
00042     ASSERT_VULKAN(result, "Failed to create an image view!")
00043 }
00044
00045 void VulkanImageView::cleanUp() {
00046     vkDestroyImageView(device->getLogicalDevice(), imageView, nullptr);
00047 }
00048
00049 VulkanImageView::~VulkanImageView() {}
```

## 6.217 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanInstance.cpp File Reference

### 6.218 VulkanInstance.cpp

[Go to the documentation of this file.](#)

```
00001 #include "VulkanInstance.hpp"
00002
00003 #include <string.h>
00004
00005 #include <string>
00006
00007 VulkanInstance::VulkanInstance() {
00008     if (ENABLE_VALIDATION_LAYERS && !check_validation_layer_support()) {
```

```

00009     throw std::runtime_error("Validation layers requested, but not available!");
00010 }
00011
00012 // info about app
00013 // most data doesn't affect program; is for developer convenience
00014 VkApplicationInfo app_info{};
00015 app_info.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
00016 app_info.pApplicationName =
00017     "\\\_/\ Epic Graphics from hell \\\_/";
00018 app_info.applicationVersion =
00019     VK_MAKE_VERSION(1, 3, 1);
00020 app_info.pEngineName = "Cataglyphis Renderer";
00021 app_info.engineVersion = VK_MAKE_VERSION(1, 3, 3);
00022 app_info.apiVersion = VK_API_VERSION_1_3;
00023
00024 // creation info for a VkInstance
00025 VkInstanceCreateInfo create_info{};
00026 create_info.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
00027 create_info.pApplicationInfo = &app_info;
00028
00029 // add validation layers IF enabled to the create info struct
00030 if (ENABLE_VALIDATION_LAYERS) {
00031     create_info.enabledLayerCount =
00032         static_cast<uint32_t>(validationLayers.size());
00033     create_info.ppEnabledLayerNames = validationLayers.data();
00034 }
00035 } else {
00036     create_info.enabledLayerCount = 0;
00037     create_info.pNext = nullptr;
00038 }
00039
00040 // create list to hold instance extensions
00041 std::vector<const char*> instance_extensions = std::vector<const char*>();
00042
00043 // Setup extensions the instance will use
00044 uint32_t glfw_extensions_count = 0; // GLFW may require multiple extensions
00045 const char** glfw_extensions; // Extensions passed as array of cstrings, so
00046 // need pointer(array) to pointer
00047
00048 // set GLFW extensions
00049 glfw_extensions = glfwGetRequiredInstanceExtensions(&glfw_extensions_count);
00050
00051 // Add GLFW extensions to list of extensions
00052 for (size_t i = 0; i < glfw_extensions_count; i++) {
00053     instance_extensions.push_back(glfw_extensions[i]);
00054 }
00055
00056 if (ENABLE_VALIDATION_LAYERS) {
00057     instance_extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
00058 }
00059
00060 // check instance extensions supported
00061 if (!check_instance_extension_support(&instance_extensions)) {
00062     throw std::runtime_error(
00063         "Vulkan instance does not support required extensions!");
00064 }
00065
00066 create_info.enabledExtensionCount =
00067     static_cast<uint32_t>(instance_extensions.size());
00068 create_info.ppEnabledExtensionNames = instance_extensions.data();
00069
00070 // create instance
00071 VkResult result = vkCreateInstance(&create_info, nullptr, &instance);
00072 ASSERT_VULKAN(result, "Failed to create a Vulkan instance!");
00073 }
00074
00075 bool VulkanInstance::check_validation_layer_support() {
00076     uint32_t layerCount;
00077     vkEnumerateInstanceLayerProperties(&layerCount, nullptr);
00078
00079     std::vector<VkLayerProperties> availableLayers(layerCount);
00080     vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());
00081
00082     for (const char* layerName : validationLayers) {
00083         bool layerFound = false;
00084
00085         for (const auto& layerProperties : availableLayers) {
00086             if (strcmp(layerName, layerProperties.layerName) == 0) {
00087                 layerFound = true;
00088                 break;
00089             }
00090         }
00091
00092         if (!layerFound) {
00093             return false;
00094         }
00095     }

```

```

00096
00097     return true;
00098 }
00099
00100 bool VulkanInstance::check_instance_extension_support(
00101     std::vector<const char*>* check_extensions) {
00102     // Need to get number of extensions to create array of correct size to hold
00103     // extensions
00104     uint32_t extension_count = 0;
00105     vkEnumerateInstanceExtensionProperties(nullptr, &extension_count, nullptr);
00106
00107     // create a list of VkExtensionProperties using count
00108     std::vector<VkExtensionProperties> extensions(extension_count);
00109     vkEnumerateInstanceExtensionProperties(nullptr, &extension_count,
00110                                         extensions.data());
00111
00112     // check if given extensions are in list of available extensions
00113     for (const auto& check_extension : *check_extensions) {
00114         bool has_extension = false;
00115
00116         for (const auto& extension : extensions) {
00117             if (strcmp(check_extension, extension.extensionName)) {
00118                 has_extension = true;
00119                 break;
00120             }
00121         }
00122
00123         if (!has_extension) {
00124             return false;
00125         }
00126     }
00127
00128     return true;
00129 }
00130
00131 void VulkanInstance::cleanUp() { vkDestroyInstance(instance, nullptr); }
00132
00133 VulkanInstance::~VulkanInstance() {}

```

## 6.219 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkan\_base/VulkanSwapChain.cpp File Reference

### 6.220 VulkanSwapChain.cpp

[Go to the documentation of this file.](#)

```

00001 #include "VulkanSwapChain.hpp"
00002
00003 #include <limits>
00004
00005 #include "Utilities.hpp"
00006
00007 VulkanSwapChain::VulkanSwapChain() {}
00008
00009 void VulkanSwapChain::initVulkanContext(VulkanDevice* device, Window* window,
00010                                         const VkSurfaceKHR& surface) {
00011     this->device = device;
00012     this->window = window;
00013
00014     // get swap chain details so we can pick the best settings
00015     SwapChainDetails swap_chain_details = device->getSwapchainDetails();
00016
00017     // 1. choose best surface format
00018     // 2. choose best presentation mode
00019     // 3. choose swap chain image resolution
00020
00021     VkSurfaceFormatKHR surface_format =
00022         choose_best_surface_format(swap_chain_details.formats);
00023     VkPresentModeKHR present_mode =
00024         choose_best_presentation_mode(swap_chain_details.presentation_mode);
00025     VkExtent2D extent =
00026         choose_swap_extent(swap_chain_details.surface_capabilities);
00027
00028     // how many images are in the swap chain; get 1 more than the minimum to allow
00029     // triple buffering
00030     uint32_t image_count =
00031         swap_chain_details.surface_capabilities.minImageCount + 1;
00032

```

```

00033 // if maxImageCount == 0, then limitless
00034 if (swap_chain_details.surface_capabilities.maxImageCount > 0 &&
00035     swap_chain_details.surface_capabilities.maxImageCount < image_count) {
00036     image_count = swap_chain_details.surface_capabilities.maxImageCount;
00037 }
00038
00039 VkSwapchainCreateInfoKHR swap_chain_create_info{};
00040 swap_chain_create_info.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
00041 swap_chain_create_info.surface = surface; // swapchain surface
00042 swap_chain_create_info.imageFormat =
00043     surface_format.format; // swapchain format
00044 swap_chain_create_info.imageColorSpace =
00045     surface_format.colorSpace; // swapchain color space
00046 swap_chain_create_info.presentMode =
00047     present_mode; // swapchain presentation mode
00048 swap_chain_create_info.imageExtent = extent; // swapchain image extents
00049 swap_chain_create_info.minImageCount =
00050     image_count; // minimum images in swapchain
00051 swap_chain_create_info.imageArrayLayers =
00052     1; // number of layers for each image in chain
00053 swap_chain_create_info.imageUsage =
00054     VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT |
00055     VK_IMAGE_USAGE_STORAGE_BIT |
00056     VK_IMAGE_USAGE_TRANSFER_DST_BIT; // what attachment images will be used
00057     // as
00058 swap_chain_create_info.preTransform =
00059     swap_chain_details.surface_capabilities
00060     .currentTransform; // transform to perform on swap chain images
00061 swap_chain_create_info.compositeAlpha =
00062     VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR; // dont do blending; everything opaque
00063 swap_chain_create_info.clipped = VK_TRUE; // of course activate clipping ! :)
00064
00065 // get queue family indices
00066 QueueFamilyIndices indices = device->getQueueFamilies();
00067
00068 // if graphics and presentation families are different then swapchain must let
00069 // images be shared between families
00070 if (indices.graphics_family != indices.presentation_family) {
00071     uint32_t queue_family_indices[] = {(uint32_t)indices.graphics_family,
00072                                         (uint32_t)indices.presentation_family};
00073
00074     swap_chain_create_info.imageSharingMode =
00075         VK_SHARING_MODE_CONCURRENT; // image share handling
00076     swap_chain_create_info.queueFamilyIndexCount =
00077         2; // number of queues to share images between
00078     swap_chain_create_info.pQueueFamilyIndices =
00079         queue_family_indices; // array of queues to share between
00080
00081 } else {
00082     swap_chain_create_info.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
00083     swap_chain_create_info.queueFamilyIndexCount = 0;
00084     swap_chain_create_info.pQueueFamilyIndices = nullptr;
00085 }
00086
00087 // if old swap chain been destroyed and this one replaces it then link old one
00088 // to quickly hand over responsibilities
00089 swap_chain_create_info.oldSwapchain = VK_NULL_HANDLE;
00090
00091 // create swap chain
00092 VkResult result = vkCreateSwapchainKHR(
00093     device->getLogicalDevice(), &swap_chain_create_info, nullptr, &swapchain);
00094 ASSERT_VULKAN(result, "Failed create swapchain!");
00095
00096 // store for later reference
00097 swap_chain_image_format = surface_format.format;
00098 swap_chain_extent = extent;
00099
00100 // get swapchain images (first count, then values)
00101 uint32_t swapchain_image_count;
00102 vkGetSwapchainImagesKHR(device->getLogicalDevice(), swapchain,
00103     &swapchain_image_count, nullptr);
00104 std::vector<VkImage> images(swapchain_image_count);
00105 vkGetSwapchainImagesKHR(device->getLogicalDevice(), swapchain,
00106     &swapchain_image_count, images.data());
00107
00108 swap_chain_images.clear();
00109
00110 for (size_t i = 0; i < images.size(); i++) {
00111     VkImage image = images[static_cast<uint32_t>(i)];
00112     // store image handle
00113     Texture swap_chain_image();
00114     swap_chain_image.setImage(image);
00115     swap_chain_image.createImageView(device, swap_chain_image_format,
00116                                     VK_IMAGE_ASPECT_COLOR_BIT, 1);
00117
00118     // add to swapchain image list
00119     swap_chain_images.push_back(swap_chain_image);

```

```

00120     }
00121 }
00122
00123 void VulkanSwapChain::cleanUp() {
00124     for (Texture& image : swap_chain_images) {
00125         vkDestroyImageView(device->getLogicalDevice(), image.getImageView(),
00126                             nullptr);
00127     }
00128
00129     vkDestroySwapchainKHR(device->getLogicalDevice(), swapchain, nullptr);
00130 }
00131
00132 VulkanSwapChain::~VulkanSwapChain() {}
00133
00134 VkSurfaceFormatKHR VulkanSwapChain::choose_best_surface_format(
00135     const std::vector<VkSurfaceFormatKHR>& formats) {
00136     // best format is subjective, but I go with:
00137     // Format:           VK_FORMAT_R8G8B8A8_UNORM (backup-format:
00138     //   VK_FORMAT_B8G8R8A8_UNORM) color_space:  VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
00139     //   the condition in if means all formats are available (no restrictions)
00140     if (formats.size() == 1 && formats[0].format == VK_FORMAT_UNDEFINED) {
00141         return {VK_FORMAT_R8G8B8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR};
00142     }
00143
00144     // if restricted, search for optimal format
00145     for (const auto& format : formats) {
00146         if ((format.format == VK_FORMAT_R8G8B8A8_UNORM ||
00147              format.format == VK_FORMAT_B8G8R8A8_UNORM) &&
00148              format.colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
00149             return format;
00150         }
00151     }
00152
00153     // in case just return first one--- but really shouldn't be the case ....
00154     return formats[0];
00155 }
00156
00157 VkPresentModeKHR VulkanSwapChain::choose_best_presentation_mode(
00158     const std::vector<	VkPresentModeKHR>& presentation_modes) {
00159     // look for mailbox presentation mode
00160     for (const auto& presentation_mode : presentation_modes) {
00161         if (presentation_mode == VK_PRESENT_MODE_MAILBOX_KHR) {
00162             return presentation_mode;
00163         }
00164     }
00165
00166     // if can't find, use FIFO as Vulkan spec says it must be present
00167     return VK_PRESENT_MODE_FIFO_KHR;
00168 }
00169
00170 VkExtent2D VulkanSwapChain::choose_swap_extent(
00171     const VkSurfaceCapabilitiesKHR& surface_capabilities) {
00172     // if current extent is at numeric limits, than extent can vary. Otherwise it
00173     // is size of window
00174     if (surface_capabilities.currentExtent.width !=
00175         std::numeric_limits<uint32_t>::max()) {
00176         return surface_capabilities.currentExtent;
00177     } else {
00178         int width, height;
00179         glfwGetFramebufferSize(window->get_window(), &width, &height);
00180
00181         // create new extent using window size
00182         VkExtent2D new_extent{};
00183         new_extent.width = static_cast<uint32_t>(width);
00184         new_extent.height = static_cast<uint32_t>(height);
00185
00186         // surface also defines max and min, so make sure within boundaries bly
00187         // clamping value
00188         new_extent.width = std::max(
00189             surface_capabilities.minImageExtent.width,
00190             std::min(surface_capabilities.maxImageExtent.width, new_extent.width));
00191         new_extent.height =
00192             std::max(surface_capabilities.minImageExtent.height,
00193                     std::min(surface_capabilities.maxImageExtent.height,
00194                             new_extent.height));
00195
00196         return new_extent;
00197     }
00198 }
00199 }
```

## 6.221 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Src/window/Window.cpp File Reference

### Functions

- static void [onErrorCallback](#) (int error, const char \*description)

#### 6.221.1 Function Documentation

##### 6.221.1.1 [onErrorCallback\(\)](#)

```
static void onErrorCallback (
    int error,
    const char * description ) [static]
```

Definition at line 10 of file [Window.cpp](#).

```
00010
00011     fprintf(stderr, "GLFW Error %d: %s\n", error, description);
00012 }
```

Referenced by [Window::initialize\(\)](#).

Here is the caller graph for this function:



## 6.222 Window.cpp

[Go to the documentation of this file.](#)

```
00001 #include "Window.hpp"
00002
00003 #include <imgui.h>
00004 #include <imgui_impl_glfw.h>
00005 #include <imgui_impl_vulkan.h>
00006
00007 #include <stdexcept>
00008
00009 // GLFW Callback functions
00010 static void onErrorCallback(int error, const char* description) {
00011     fprintf(stderr, "GLFW Error %d: %s\n", error, description);
00012 }
00013
00014 Window::Window()
00015     :
00016         window_width(800.f),
00017         window_height(600.f),
00018         x_change(0.0f),
00019         y_change(0.0f),
00020         framebuffer_resized(false)
```

```

00022
00023 {
00024     // all keys non-pressed in the beginning
00025     for (size_t i = 0; i < 1024; i++) {
00026         keys[i] = 0;
00027     }
00028
00029     initialize();
00030 }
00031
00032 // please use this constructor; never the standard
00033 Window::Window(uint32_t window_width, uint32_t window_height)
00034 :
00035
00036     window_width(window_width),
00037     window_height(window_height),
00038     x_change(0.0f),
00039     y_change(0.0f),
00040     framebuffer_resized(false)
00041
00042 {
00043     // all keys non-pressed in the beginning
00044     for (size_t i = 0; i < 1024; i++) {
00045         keys[i] = 0;
00046     }
00047
00048     initialize();
00049 }
00050
00051 int Window::initialize() {
00052     glfwSetErrorCallback(onErrorCallback);
00053     if (!glfwInit()) {
00054         printf("GLFW Init failed!");
00055         glfwTerminate();
00056         return 1;
00057     }
00058
00059     if (!glfwVulkanSupported()) {
00060         throw std::runtime_error("No Vulkan Supported!");
00061     }
00062
00063     // allow it to resize
00064     glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
00065
00066     // retrieve new window
00067     glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
00068     main_window =
00069         glfwCreateWindow(window_width, window_height,
00070                         "\\__/_ Epic graphics from hell \\__/", NULL, NULL);
00071
00072     if (!main_window) {
00073         printf("GLFW Window creation failed!");
00074         glfwTerminate();
00075         return 1;
00076     }
00077
00078     // get buffer size information
00079     glfwGetFramebufferSize(main_window, &window_buffer_width,
00080                           &window_buffer_height);
00081
00082     init_callbacks();
00083
00084     return 0;
00085 }
00086
00087 void Window::cleanUp() {
00088     glfwDestroyWindow(main_window);
00089     glfwTerminate();
00090 }
00091
00092 void Window::update_viewport() {
00093     glfwGetFramebufferSize(main_window, &window_buffer_width,
00094                           &window_buffer_height);
00095 }
00096
00097 void Window::set_buffer_size(float window_buffer_width,
00098                               float window_buffer_height) {
00099     this->window_buffer_width = window_buffer_width;
00100     this->window_buffer_height = window_buffer_height;
00101 }
00102
00103 float Window::get_x_change() {
00104     float the_change = x_change;
00105     x_change = 0.0f;
00106     return the_change;
00107 }
00108

```

```
00109 float Window::get_y_change() {
00110     float the_change = y_change;
00111     y_change = 0.0f;
00112     return the_change;
00113 }
00114
00115 float Window::get_height() { return float(window_height); }
00116
00117 float Window::get_width() { return float(window_width); }
00118
00119 bool Window::framebuffer_size_has_changed() { return framebuffer_resized; }
00120
00121 void Window::init_callbacks() {
00122     // TODO: remember this section for our later game logic
00123     // for the space ship to fly around
00124     glfwSetWindowUserPointer(main_window, this);
00125     glfwSetKeyCallback(main_window, &key_callback);
00126     glfwSetMouseButtonCallback(main_window, &mouse_button_callback);
00127     glfwSetFrameBufferSizeCallback(main_window, &framebuffer_size_callback);
00128 }
00129
00130 void Window::framebuffer_size_callback(GLFWwindow* window, int width,
00131                                         int height) {
00132     auto app = reinterpret_cast<Window*>(glfwGetWindowUserPointer(window));
00133     app->framebuffer_resized = true;
00134     app->window_width = width;
00135     app->window_height = height;
00136 }
00137
00138 void Window::reset_framebuffer_has_changed() {
00139     this->framebuffer_resized = false;
00140 }
00141
00142 void Window::key_callback(GLFWwindow* window, int key, int code, int action,
00143                           int mode) {
00144     Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));
00145
00146     if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) {
00147         glfwSetWindowShouldClose(window, VK_TRUE);
00148     }
00149
00150     if (key >= 0 && key < 1024) {
00151         if (action == GLFW_PRESS) {
00152             the_window->keys[key] = true;
00153
00154         } else if (action == GLFW_RELEASE) {
00155             the_window->keys[key] = false;
00156         }
00157     }
00158 }
00159
00160 void Window::mouse_callback(GLFWwindow* window, double x_pos, double y_pos) {
00161     Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));
00162
00163     // need to handle first occurrence of a mouse moving event
00164     if (the_window->mouse_first_moved) {
00165         the_window->last_x = static_cast<float>(x_pos);
00166         the_window->last_y = static_cast<float>(y_pos);
00167         the_window->mouse_first_moved = false;
00168     }
00169
00170     the_window->x_change = static_cast<float>((x_pos - the_window->last_x));
00171     // take care of correct subtraction :
00172     the_window->y_change = static_cast<float>((the_window->last_y - y_pos));
00173
00174     // update params
00175     the_window->last_x = static_cast<float>(x_pos);
00176     the_window->last_y = static_cast<float>(y_pos);
00177 }
00178
00179 void Window::mouse_button_callback(GLFWwindow* window, int button, int action,
00180                                   int mods) {
00181     if (ImGui::GetCurrentContext() != nullptr &&
00182         ImGui::GetIO().WantCaptureMouse) {
00183         ImGuiIO& io = ImGui::GetIO();
00184         io.AddMouseEvent(button, action);
00185         return;
00186     }
00187
00188     Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));
00189
00190     if ((action == GLFW_PRESS) && (button == GLFW_MOUSE_BUTTON_RIGHT)) {
00191         glfwSetCursorPosCallback(window, mouse_callback);
00192     } else {
00193         the_window->mouse_first_moved = true;
00194         glfwSetCursorPosCallback(window, NULL);
00195     }
}
```

```
00196 }
00197
00198 Window::~Window() { }
```

## 6.223 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Test/commit/cmake/SetTestFilters.cmake File Reference

### 6.224 SetTestFilters.cmake

[Go to the documentation of this file.](#)

```
00001 set(RENDERER_COMMIT_TEST_FILTER           "commitSuite.cpp")
00002
00003 source_group("COMMIT_TEST_FILES"          FILES ${RENDERER_COMMIT_TEST_FILTER})
```

## 6.225 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Test/compile/cmake/SetTestFilters.cmake File Reference

### 6.226 SetTestFilters.cmake

[Go to the documentation of this file.](#)

```
00001 set(RENDERER_COMPILE_TEST_FILTER          "compileSuite.cpp")
00002
00003 source_group("COMPILE_TEST_FILES"          FILES ${RENDERER_COMPILE_TEST_FILTER})
```

## 6.227 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Test/commit/CMakeLists.txt File Reference

### Functions

- `set` (WORKING\_DIRECTORY \${CMAKE\_CURRENT\_SOURCE\_DIR}../../) `set`(PROJECT\_SRC\_DIR \$
- Src `set` (EXTERNAL\_LIB\_SRC\_DIR \${WORKING\_DIRECTORY}ExternalLib) `set`(SHADER\_SRC\_DIR \$
- Resources Shaders `include` (\${WORKING\_DIRECTORY}cmake/filters/SetShaderFilters.cmake) `include`(\$

#### 6.227.1 Function Documentation

##### 6.227.1.1 `include()`

```
cmake filters SetProjectFilters cmake include (
    ${WORKING_DIRECTORY}cmake/filters/SetShaderFilters. )
```

Definition at line 10 of file [CMakeLists.txt](#).

```
00010     ${WORKING_DIRECTORY}cmake/filters/SetShaderFilters.cmake)
00011 include(${WORKING_DIRECTORY}cmake/filters/SetProjectFilters.cmake)
```

### 6.227.1.2 set() [1/2]

```
Src set (
    EXTERNAL_LIB_SRC_DIR ${WORKING_DIRECTORY}ExternalLib/ )
```

Definition at line 7 of file [CMakeLists.txt](#).

```
00007           ${WORKING_DIRECTORY}ExternalLib/
00008 set (SHADER_SRC_DIR ${WORKING_DIRECTORY}Resources/Shaders/)
```

### 6.227.1.3 set() [2/2]

```
set (
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/.../... )
```

Definition at line 3 of file [CMakeLists.txt](#).

```
00003           ${CMAKE_CURRENT_SOURCE_DIR}/.../...
00004
00005 # update current positions
00006 set (PROJECT_SRC_DIR ${WORKING_DIRECTORY}Src/)
```

## 6.228 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Test/compile/CMakeLists.txt File Reference

### Functions

- `set (WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/.../...) set(PROJECT_SRC_DIR $`
- Src `set (EXTERNAL_LIB_SRC_DIR ${WORKING_DIRECTORY}ExternalLib/) set(SHADER_SRC_DIR $`
- Resources Shaders `include (${WORKING_DIRECTORY}cmake/filters/SetShaderFilters.cmake) include($`

### 6.228.1 Function Documentation

#### 6.228.1.1 include()

```
Resources Shaders include (
    ${WORKING_DIRECTORY}cmake/filters/SetShaderFilters. )
```

Definition at line 8 of file [CMakeLists.txt](#).

```
00008           ${WORKING_DIRECTORY}cmake/filters/SetShaderFilters.cmake)
00009 include (${WORKING_DIRECTORY}cmake/filters/SetProjectFilters.cmake)
```

### 6.228.1.2 set() [1/2]

```
Src set (
    EXTERNAL_LIB_SRC_DIR ${WORKING_DIRECTORY}ExternalLib/ )
```

Definition at line 5 of file [CMakeLists.txt](#).

```
00005           ${WORKING_DIRECTORY}ExternalLib/
00006 set (SHADER_SRC_DIR ${WORKING_DIRECTORY}Resources/Shaders/)
```

### 6.228.1.3 set() [2/2]

```
set (
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/.../... )
```

Definition at line 1 of file [CMakeLists.txt](#).

```
00001           ${CMAKE_CURRENT_SOURCE_DIR}/.../...
00002
00003 # update current positions
00004 set (PROJECT_SRC_DIR ${WORKING_DIRECTORY}Src/)
```

## 6.229 C:/Users/jonas\_I6e3q/Desktop/GraphicsEngineVulkan/← Test/commit/commitSuite.cpp File Reference

### Functions

- [TEST](#) (HelloTest1, BasicAssertions)
- [TEST](#) (VulkanBuffer1, blob)

### 6.229.1 Function Documentation

#### 6.229.1.1 TEST() [1/2]

```
TEST (
    HelloTest1 ,
    BasicAssertions )
```

Definition at line 5 of file [commitSuite.cpp](#).

```
00005
00006
00007     // Expect two strings not to be equal.
00008     EXPECT_STRNE("hello", "world");
00009     // Expect equality.
00010     EXPECT_EQ(7 * 6, 42);
00011
00012 }
```

### 6.229.1.2 TEST() [2/2]

```
TEST (
    VulkanBuffer1 ,
    blob )
```

Definition at line 14 of file [commitSuite.cpp](#).

```
00015 {
00016
00017     VulkanBuffer vulkanBuffer;
00018
00019     int c = 0;
00020
00021     // Test that counter 0 returns 0
00022     EXPECT_EQ(0, c);
00023
00024     // EXPECT_EQ() evaluates its arguments exactly once, so they
00025     // can have side effects.
00026
00027     EXPECT_EQ(0, c++);
00028     EXPECT_EQ(1, c++);
00029     EXPECT_EQ(2, c++);
00030
00031     EXPECT_EQ(3, c++);
00032 }
```

## 6.230 commitSuite.cpp

[Go to the documentation of this file.](#)

```
00001 #include <gtest/gtest.h>
00002 #include "VulkanBuffer.hpp"
00003
00004 // Demonstrate some basic assertions.
00005 TEST(HelloTest1, BasicAssertions) {
00006
00007     // Expect two strings not to be equal.
00008     EXPECT_STRNE("hello", "world");
00009     // Expect equality.
00010     EXPECT_EQ(7 * 6, 42);
00011
00012 }
00013
00014 TEST(VulkanBuffer1, blob)
00015 {
00016
00017     VulkanBuffer vulkanBuffer;
00018
00019     int c = 0;
00020
00021     // Test that counter 0 returns 0
00022     EXPECT_EQ(0, c);
00023
00024     // EXPECT_EQ() evaluates its arguments exactly once, so they
00025     // can have side effects.
00026
00027     EXPECT_EQ(0, c++);
00028     EXPECT_EQ(1, c++);
00029     EXPECT_EQ(2, c++);
00030
00031     EXPECT_EQ(3, c++);
00032 }
```

## 6.231 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/← Test/compile/compileSuite.cpp File Reference

### Functions

- [TEST \(HelloTest2, BasicAssertions\)](#)
- [TEST \(VulkanBuffer2, blob\)](#)

## 6.231.1 Function Documentation

### 6.231.1.1 TEST() [1/2]

```
TEST (
    HelloTest2 ,
    BasicAssertions )
```

Definition at line 5 of file [compileSuite.cpp](#).

```
00005 {
00006
00007     // Expect two strings not to be equal.
00008     EXPECT_STRNE("hello", "world");
00009     // Expect equality.
00010     EXPECT_EQ(7 * 6, 42);
00011
00012 }
```

### 6.231.1.2 TEST() [2/2]

```
TEST (
    VulkanBuffer2 ,
    blob )
```

Definition at line 14 of file [compileSuite.cpp](#).

```
00015 {
00016
00017     VulkanBuffer vulkanBuffer;
00018
00019     int c = 0;
00020
00021     // Test that counter 0 returns 0
00022     EXPECT_EQ(0, c);
00023
00024     // EXPECT_EQ() evaluates its arguments exactly once, so they
00025     // can have side effects.
00026
00027     EXPECT_EQ(0, c++);
00028     EXPECT_EQ(1, c++);
00029     EXPECT_EQ(2, c++);
00030
00031     EXPECT_EQ(3, c++);
00032 }
```

## 6.232 compileSuite.cpp

[Go to the documentation of this file.](#)

```
00001 #include <gtest/gtest.h>
00002 #include "VulkanBuffer.hpp"
00003
00004 // Demonstrate some basic assertions.
00005 TEST(HelloTest2, BasicAssertions) {
00006
00007     // Expect two strings not to be equal.
00008     EXPECT_STRNE("hello", "world");
00009     // Expect equality.
00010     EXPECT_EQ(7 * 6, 42);
00011
00012 }
00013
00014 TEST(VulkanBuffer2, blob)
```

```
00015 {
00016
00017     VulkanBuffer vulkanBuffer;
00018
00019     int c = 0;
00020
00021     // Test that counter 0 returns 0
00022     EXPECT_EQ(0, c);
00023
00024     // EXPECT_EQ() evaluates its arguments exactly once, so they
00025     // can have side effects.
00026
00027     EXPECT_EQ(0, c++);
00028     EXPECT_EQ(1, c++);
00029     EXPECT_EQ(2, c++);
00030
00031     EXPECT_EQ(3, c++);
00032 }
```



# Index

~ASManager  
    ASManager, 23

~Allocator  
    Allocator, 18

~App  
    App, 20

~Camera  
    Camera, 45

~CommandBufferManager  
    CommandBufferManager, 54

~File  
    File, 58

~GUI  
    GUI, 63

~Mesh  
    Mesh, 80

~Model  
    Model, 93

~PathTracing  
    PathTracing, 114

~PostStage  
    PostStage, 126

~Rasterizer  
    Rasterizer, 151

~Raytracing  
    Raytracing, 170

~Scene  
    Scene, 185

~ShaderHelper  
    ShaderHelper, 199

~Texture  
    Texture, 207

~VulkanBuffer  
    VulkanBuffer, 226

~VulkanBufferManager  
    VulkanBufferManager, 233

~VulkanDevice  
    VulkanDevice, 240

~VulkanImage  
    VulkanImage, 257

~VulkanImageView  
    VulkanImageView, 267

~VulkanInstance  
    VulkanInstance, 273

~VulkanRenderer  
    VulkanRenderer, 280

~VulkanSwapChain  
    VulkanSwapChain, 325

~Window

Window, 338

accessFlagsForImageLayout  
    VulkanImage, 257

add\_model  
    Scene, 186

add\_new\_mesh  
    Model, 93

add\_object\_description  
    Scene, 186

addSampler  
    Model, 93

addTexture  
    Model, 94

align\_up  
    MemoryHelper.hpp, 361

Allocator, 17

    ~Allocator, 18

    Allocator, 18

    cleanUp, 19

    vmaAllocator, 19

allocator  
    VulkanRenderer, 315

ambient  
    ObjMaterial, 110

App, 20

    ~App, 20

    App, 20

    run, 21

as\_build\_offset\_info  
    BlasInput, 38

as\_geometry  
    BlasInput, 38

ASManager, 22

    ~ASManager, 23

    ASManager, 23

    blas, 36

    cleanUp, 23

    commandBufferManager, 36

    createAccelerationStructureInfosBLAS, 24

    createASForScene, 26

    createBLAS, 27

    createSingleBlas, 29

    createTLAS, 30

    getTLAS, 34

    objectToVkGeometryKHR, 34

    tlas, 36

    vulkanBufferManager, 37

    vulkanDevice, 37

asManager

VulkanRenderer, 316  
**aspect\_ratio**  
 PushConstantPost, 145

**beginCommandBuffer**  
 CommandBufferManager, 55

**blas**  
 ASManager, 36

**BlasInput**, 37  
 as\_build\_offset\_info, 38  
 as\_geometry, 38

**BottomLevelAccelerationStructure**, 39  
 vulkanAS, 40  
 vulkanBuffer, 40

**buffer**  
 VulkanBuffer, 230

**bufferMemory**  
 VulkanBuffer, 230

**build\_info**  
 BuildAccelerationStructure, 41

**BuildAccelerationStructure**, 40  
 build\_info, 41  
 range\_info, 42  
 single\_bla, 42  
 size\_info, 42

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/CompilerWarnings.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/Ra  
 351  
 378

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/CompileShadersToSPV.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/Ra  
 352  
 378, 379

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/Doxygen.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/S  
 353  
 379, 381

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetExternalLibsFilters.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/S  
 353  
 381

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetProjectFilters.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/V  
 354  
 381, 382

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/filters/SetShaderFilters.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/V  
 356  
 383

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/Sanitizers.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/renderer/V  
 357  
 384

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/SetSourceGroups.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Cam  
 357  
 384

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/cmake/StaticAnalyzers.cmake  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/GUI  
 358  
 385

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/app/App.hpp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Mesh  
 358  
 385, 386

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/common/FormatHelper.hpp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Mod  
 358, 359  
 386, 387

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/common/Globals.hpp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Obj  
 360  
 387

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/common/MemoryHelper.hpp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ObjL  
 361, 362  
 388

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/common/Utilities.hpp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/ObjM  
 362, 363  
 388, 390

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/gui/GUI.hpp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Scen  
 363  
 390

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/memory/Allocator.hpp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/include/scene/Scen  
 364  
 391, 392

C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">Textures.h</a>	410
392		418
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">Vert.h</a>	410
393		418
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">Ural.h</a>	410
394		419
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">Shaders.h</a>	410
394, 395		419
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">UralLight.h</a>	410
395		421
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">UralMaterial.h</a>	410
396		421
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">UralTexture.h</a>	410
397		422
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">UralVertex.h</a>	410
398		422
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">UralWorld.h</a>	410
399		423
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">UralWorldLight.h</a>	410
399, 400		423
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Resources/Shaders/</a>	<a href="#">UralWorldMaterial.h</a>	410
400		423
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Src/app/</a>	<a href="#">App.cpp</a>	401
401		423
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/include/ <a href="#">GraphicsEngineVulkan/Src/gui/</a>	<a href="#">GUI.cpp</a>	401, 402
402		424
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Main.cpp</a>	402
402		427, 428
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/MemoryAlloc.cpp</a>	405
405		428
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Renderer/acceleration.h</a>	406
406		428
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Renderer/CommandBuffer.h</a>	407
407		435
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Renderer/PathTracing.h</a>	408
408		436
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Renderer/PostProcessing.h</a>	409
409		439
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Renderer/Rasterizer.h</a>	410
410		445
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Renderer/Raytracer.h</a>	410
410		452
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/Renderer/Vulkan.h</a>	410
410		456
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/ServiceEngineVulkan.h</a>	411, 412
411, 412		470
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/SceneEngine.h</a>	412
412		472
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/SceneModel.h</a>	415
415		473
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/SceneObjLoader.h</a>	416
416		475
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/SceneScene.h</a>	416
416		477
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/SceneSceneConfig.h</a>	416
416		478
C:/Users/jonas_l6e3q/Desktop/GraphicsEngineVulkan/Resources/Shaders/	<a href="#">Basic/Desktop/SceneTexture.h</a>	416
416		479

C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/scene/[camera.cpp](#), 48  
     482  
     movement\_speed, 51  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/util/[File.cpp](#), 52  
     483  
     head\_plane, 52  
     pitch, 52  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[ShaderHelper.cpp](#), 52  
     484  
     right, 52  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanBuffer.cpp](#), 49  
     485  
     set\_far\_plane, 49  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanBufferManager.cpp](#),  
     486  
     set\_near\_plane, 50  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanBufferPool.cpp](#), 52  
     487  
     Debug.cpp,  
     up, 53  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanDevice.cpp](#),  
     488  
     world\_up, 53  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanImage.cpp](#),  
     493  
     check\_device\_extension\_support  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanDeviceView.cpp](#),  
     496  
     check\_device\_suitable  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanDeviceInstance.cpp](#),  
     496  
     check\_instance\_extension\_support  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/vulkanbase/[VulkanSwapChain.cpp](#),  
     498  
     check\_validation\_layer\_support  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Src/window/[Window.cpp](#), 274  
     501  
     checkChangedFrameBufferSize  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Test/compute/[VulkanComputeSuite.cpp](#), 281  
     504  
     choose\_best\_presentation\_mode  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Test/compute/[VulkanComputeSuite.h](#), 325  
     504  
     choose\_best\_surface\_format  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Test/compute/[VulkanComputeCheck.cpp](#), 326  
     506, 507  
     choose\_supported\_format  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Test/compute/[VulkanComputeTest.cpp](#), 338  
     504  
     choose\_swap\_extent  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Test/compute/[VulkanComputeCheck.h](#), 326  
     505  
     cleanUp  
 C:/Users/jonas\_l6e3q/Desktop/GraphicsEngineVulkan/Test/compute/[AllocationSuite.cpp](#),  
     507, 508  
     ASManager, 23  
 calculate\_viewmatrix  
     Camera, 45  
 call\_region  
     Raytracing, 180  
 cam\_pos  
     SceneUBO, 197  
 Camera, 43  
     ~Camera, 45  
     calculate\_viewmatrix, 45  
     Camera, 44  
     far\_plane, 51  
     fov, 51  
     front, 51  
     get\_camera\_direction, 45  
     get\_camera\_position, 46  
     get\_far\_plane, 46  
     get\_fov, 47  
     get\_near\_plane, 47  
     get\_right\_axis, 47  
     get\_up\_axis, 48  
     get\_yaw, 48  
     key\_control, 48  
     VulkanImage, 258  
     VulkanImage View, 267  
     VulkanInstance, 274  
     VulkanRenderer, 283  
     VulkanSwapChain, 327  
     Window, 338  
     cleanUpCommandPools  
         VulkanRenderer, 284  
     cleanUpSync  
         VulkanRenderer, 284  
     cleanUpUBOs  
         VulkanRenderer, 284

VulkanRenderer, 285  
clear\_color  
    PushConstantRaytracing, 147  
clearColor  
    PushConstantPathTracing, 144  
CMakeLists.txt  
    include, 504, 505  
    set, 504–506  
color  
    Vertex, 223  
command\_buffers  
    VulkanRenderer, 316  
command\_pool  
    ObjLoader, 108  
CommandBufferManager, 54  
    ~CommandBufferManager, 54  
    beginCommandBuffer, 55  
    CommandBufferManager, 54  
    endAndSubmitCommandBuffer, 55  
commandBufferManager  
    ASManager, 36  
    GUI, 71  
    Rasterizer, 166  
    Texture, 218  
    VulkanBufferManager, 237  
    VulkanImage, 265  
    VulkanRenderer, 316  
commitSuite.cpp  
    TEST, 506  
compileShader  
    ShaderHelper, 199  
compileSuite.cpp  
    TEST, 508  
compute\_command\_pool  
    VulkanRenderer, 316  
compute\_family  
    QueueFamilyIndices, 149  
compute\_queue  
    VulkanDevice, 253  
computeLimits  
    PathTracing, 121  
copyBuffer  
    VulkanBufferManager, 233  
copyImageBuffer  
    VulkanBufferManager, 235  
create  
    VulkanBuffer, 227  
    VulkanImage, 259  
    VulkanImageView, 268  
create\_command\_buffers  
    VulkanRenderer, 285  
create\_command\_pool  
    VulkanRenderer, 286  
create\_fonts\_and\_upload  
    GUI, 64  
create\_gui\_context  
    GUI, 65  
create\_logical\_device  
    VulkanDevice, 242  
    create\_object\_description\_buffer  
        VulkanRenderer, 287  
    create\_post\_descriptor\_layout  
        VulkanRenderer, 289  
    create\_surface  
        VulkanRenderer, 291  
    create\_uniform\_buffers  
        VulkanRenderer, 291  
createAccelerationStructureInfosBLAS  
    ASManager, 24  
createASForScene  
    ASManager, 26  
createBLAS  
    ASManager, 27  
createBufferAndUploadVectorOnDevice  
    VulkanBufferManager, 236  
created  
    VulkanBuffer, 230  
createDepthbufferImage  
    PostStage, 127  
createDescriptorPoolSharedRenderStages  
    VulkanRenderer, 292  
createFramebuffer  
    PostStage, 128  
    Rasterizer, 152  
createFromFile  
    Texture, 208  
createGraphicsPipeline  
    PostStage, 129  
    Rasterizer, 153  
    Raytracing, 171  
createImage  
    Texture, 210  
createImageView  
    Texture, 211  
createIndexBuffer  
    Mesh, 81  
createMaterialBuffer  
    Mesh, 82  
createMaterialIDBuffer  
    Mesh, 83  
createOffscreenTextureSampler  
    PostStage, 133  
createPCRange  
    Raytracing, 174  
createPipeline  
    PathTracing, 114  
createPushConstantRange  
    PostStage, 134  
    Rasterizer, 157  
createQueryPool  
    PathTracing, 116  
createRaytracingDescriptorPool  
    VulkanRenderer, 294  
createRaytracingDescriptorSetLayouts  
    VulkanRenderer, 295  
createRaytracingDescriptorSets

VulkanRenderer, 295  
 createRenderPass  
     Rasterizer, 157  
 createRenderpass  
     PostStage, 134  
 createSBT  
     Raytracing, 175  
 createShaderModule  
     ShaderHelper, 200  
 createSharedRenderDescriptorSet  
     VulkanRenderer, 296  
 createSharedRenderDescriptorSetLayouts  
     VulkanRenderer, 298  
 createSingleBlas  
     ASManager, 29  
 createSynchronization  
     VulkanRenderer, 299  
 createTextures  
     Rasterizer, 159  
 createTLAS  
     ASManager, 30  
 createVertexBuffer  
     Mesh, 84  
 current\_frame  
     VulkanRenderer, 316  
  
 debug, 9  
     debugUtilsMessenger, 12  
     debugUtilsMessengerCallback, 9  
     freeDebugCallback, 10  
     messageCallback, 11  
     setupDebugging, 11  
     validationLayerCount, 12  
     validationLayerNames, 13  
     vkCreateDebugUtilsMessengerEXT, 13  
     vkDestroyDebugUtilsMessengerEXT, 13  
 debugUtilsMessenger  
     debug, 12  
 debugUtilsMessengerCallback  
     debug, 9  
 depth\_format  
     PostStage, 141  
 depthBufferImage  
     PostStage, 141  
     Rasterizer, 166  
 descriptorPoolSharedRenderStages  
     VulkanRenderer, 317  
 device  
     GUI, 71  
     Mesh, 88  
     Model, 99  
     ObjLoader, 108  
     PathTracing, 121  
     PostStage, 141  
     Rasterizer, 167  
     Raytracing, 180  
     VulkanBuffer, 231  
     VulkanImage, 265  
     VulkanImageView, 270  
  
     VulkanRenderer, 317  
     VulkanSwapChain, 333  
 device\_extensions  
     VulkanDevice, 253  
 device\_extensions\_for\_raytracing  
     VulkanDevice, 253  
 device\_properties  
     VulkanDevice, 253  
 diffuse  
     ObjMaterial, 111  
 direcional\_light\_radiance  
     GUISceneSharedVars, 74  
 directional\_light\_color  
     GUISceneSharedVars, 74  
 directional\_light\_direction  
     GUISceneSharedVars, 75  
 dissolve  
     ObjMaterial, 111  
 drawFrame  
     VulkanRenderer, 300  
  
 emission  
     ObjMaterial, 111  
 ENABLE\_VALIDATION\_LAYERS  
     Utilities.hpp, 363  
 endAndSubmitCommandBuffer  
     CommandBufferManager, 55  
  
 far\_plane  
     Camera, 51  
 File, 57  
     ~File, 58  
     File, 57  
     file\_location, 60  
     getBaseDir, 58  
     read, 58  
     readCharSequence, 59  
 file\_location  
     File, 60  
 find\_memory\_type\_index  
     MemoryHelper.hpp, 361  
 finishAllRenderCommands  
     VulkanRenderer, 303  
 FormatHelper.hpp  
     choose\_supported\_format, 359  
 formats  
     SwapChainDetails, 205  
 fov  
     Camera, 51  
 framebuffer  
     Rasterizer, 167  
 framebuffer\_resized  
     Window, 347  
 framebuffer\_size\_callback  
     Window, 338  
 framebuffer\_size\_has\_changed  
     Window, 339  
 framebuffers  
     PostStage, 141

freeDebugCallback  
    debug, 10  
front  
    Camera, 51  
  
generateMipMaps  
    Texture, 212  
get\_buffer\_height  
    Window, 339  
get\_buffer\_width  
    Window, 339  
get\_camera\_direction  
    Camera, 45  
get\_camera\_position  
    Camera, 46  
get\_far\_plane  
    Camera, 46  
get\_fov  
    Camera, 47  
get\_height  
    Window, 340  
get\_keys  
    Window, 340  
get\_model\_list  
    Scene, 186  
get\_near\_plane  
    Camera, 47  
get\_physical\_device  
    VulkanDevice, 245  
get\_right\_axis  
    Camera, 47  
get\_should\_close  
    Window, 340  
get\_up\_axis  
    Camera, 48  
get\_width  
    Window, 340  
get\_window  
    Window, 341  
get\_x\_change  
    Window, 341  
get\_y\_change  
    Window, 341  
get\_yaw  
    Camera, 48  
getBaseDir  
    File, 58  
getBuffer  
    VulkanBuffer, 229  
getBufferMemory  
    VulkanBuffer, 229  
getComputeQueue  
    VulkanDevice, 246  
getCustomInstanceIndex  
    Model, 95  
getGraphicsQueue  
    VulkanDevice, 246  
getGuiRendererSharedVars  
    GUI, 67  
  
getGuiSceneSharedVars  
    GUI, 68  
    Scene, 187  
getImage  
    Texture, 214  
    VulkanImage, 260  
getImageView  
    Texture, 214  
    VulkanImageView, 269  
getIndexBuffer  
    Mesh, 85  
    Scene, 187  
getIndexCount  
    Mesh, 86  
    Scene, 188  
getLogicalDevice  
    VulkanDevice, 246  
getMaterialIDBuffer  
    Mesh, 86  
getMesh  
    Model, 96  
getMeshCount  
    Model, 96  
    Scene, 188  
getMipLevel  
    Texture, 215  
getModel  
    Mesh, 86  
    Model, 96  
getModelCount  
    Scene, 189  
getModelFile  
    sceneConfig, 13  
getModelMatrix  
    Scene, 189  
    sceneConfig, 14  
getNumberMeshes  
    Scene, 190  
getNumberObjectDescriptions  
    Scene, 190  
getNumberSwapChainImages  
    VulkanSwapChain, 328  
getObjectDescription  
    Mesh, 87  
    Model, 96  
getObjectDescriptions  
    Scene, 190  
getOffscreenSampler  
    PostStage, 136  
getOffscreenTexture  
    Rasterizer, 161  
getPhysicalDevice  
    VulkanDevice, 248  
getPhysicalDeviceProperties  
    VulkanDevice, 249  
getPresentationQueue  
    VulkanDevice, 249  
getPrimitiveCount

Model, 97  
 getQueueFamilies  
     VulkanDevice, 249, 250  
 getRenderPass  
     PostStage, 137  
 getShaderSpvDir  
     ShaderHelper, 201  
 getSwapChain  
     VulkanSwapChain, 329  
 getSwapchainDetails  
     VulkanDevice, 251, 252  
 getSwapChainExtent  
     VulkanSwapChain, 329  
 getSwapChainFormat  
     VulkanSwapChain, 330  
 getSwapChainImage  
     VulkanSwapChain, 330  
 getTextureCount  
     Model, 97  
     Scene, 191  
 getTextureList  
     Model, 98  
 getTextures  
     Model, 98  
     Scene, 191  
 getTextureSampler  
     Scene, 192  
 getTextureSamplers  
     Model, 98  
 getTLAS  
     ASManager, 34  
 getVertexBuffer  
     Mesh, 87  
     Scene, 192  
 getVertexCount  
     Mesh, 88  
 getVertexInputAttributeDesc  
     vertex, 15  
 getVulkanImage  
     Texture, 215  
 getVulkanImageView  
     Texture, 215  
 getVulkanInstance  
     VulkanInstance, 275  
**Globals.hpp**  
     MAX\_FRAME\_DRAWNS, 360  
     MAX\_OBJECTS, 360  
**GlobalUBO**, 60  
     projection, 61  
     view, 61  
**globalUBO**  
     VulkanRenderer, 317  
**GlobalUBO.hpp**  
     mat4, 367  
     uint, 367  
     vec2, 367  
     vec3, 368  
     vec4, 368  
 globalUBOBuffer  
     VulkanRenderer, 317  
 graphics\_command\_pool  
     VulkanRenderer, 318  
 graphics\_family  
     QueueFamilyIndices, 149  
 graphics\_pipeline  
     PostStage, 141  
     Rasterizer, 167  
 graphics\_queue  
     VulkanDevice, 254  
 graphicsPipeline  
     Raytracing, 180  
**GUI**, 62  
     ~GUI, 63  
     cleanUp, 63  
     commandBufferManager, 71  
     create\_fonts\_and\_upload, 64  
     create\_gui\_context, 65  
     device, 71  
     getGuiRendererSharedVars, 67  
     getGuiSceneSharedVars, 68  
     GUI, 63  
     gui\_descriptor\_pool, 71  
     guiRendererSharedVars, 71  
     guiSceneSharedVars, 71  
     initializeVulkanContext, 68  
     render, 69  
     window, 72  
 gui  
     VulkanRenderer, 318  
 gui\_descriptor\_pool  
     GUI, 71  
 GUIRendererSharedVars, 72  
     pathTracing, 73  
     raytracing, 73  
     shader\_hot\_reload\_triggered, 73  
 guiRendererSharedVars  
     GUI, 71  
 GUISceneSharedVars, 74  
     direcional\_light\_radiance, 74  
     directional\_light\_color, 74  
     directional\_light\_direction, 75  
 guiSceneSharedVars  
     GUI, 71  
     Scene, 195  
 height  
     PushConstantPathTracing, 144  
 hit\_region  
     Raytracing, 181  
 hitShaderBindingTableBuffer  
     Raytracing, 181  
 host\_device\_shared\_vars.hpp  
     MAX\_TEXTURE\_COUNT, 411  
 illum  
     ObjMaterial, 111  
 image

VulkanImage, 265  
image\_available  
    VulkanRenderer, 318  
imageMemory  
    VulkanImage, 265  
images\_in\_flight\_fences  
    VulkanRenderer, 318  
imageView  
    VulkanImageView, 270  
in\_flight\_fences  
    VulkanRenderer, 318  
include  
    CMakeLists.txt, 504, 505  
index\_address  
    ObjectDescription, 101  
index\_count  
    Mesh, 88  
indexBuffer  
    Mesh, 89  
indices  
    ObjLoader, 108  
init  
    PathTracing, 117  
    PostStage, 137  
    Rasterizer, 162  
    Raytracing, 176  
init\_callbacks  
    Window, 342  
initialize  
    Window, 342  
initializeVulkanContext  
    GUI, 68  
initVulkanContext  
    VulkanSwapChain, 331  
instance  
    VulkanDevice, 254  
    VulkanInstance, 276  
    VulkanRenderer, 319  
ior  
    ObjMaterial, 111  
is\_valid  
    QueueFamilyIndices, 148  
key\_callback  
    Window, 343  
key\_control  
    Camera, 48  
keys  
    Window, 347  
last\_x  
    Window, 347  
last\_y  
    Window, 347  
light\_dir  
    SceneUBO, 197  
loadModel  
    ObjLoader, 103  
    Scene, 193  
loadTextureData  
    Texture, 216  
loadTexturesAndMaterials  
    ObjLoader, 105  
loadVertices  
    ObjLoader, 106  
logical\_device  
    VulkanDevice, 254  
main  
    Main.cpp, 427  
Main.cpp  
    main, 427  
main\_window  
    Window, 348  
mat4  
    GlobalUBO.hpp, 367  
    ObjMaterial.hpp, 389  
    PushConstantPathTracing.hpp, 371  
    PushConstantPost.hpp, 373  
    PushConstantRasterizer.hpp, 374  
    PushConstantRayTracing.hpp, 376  
    SceneUBO.hpp, 380  
material\_address  
    ObjectDescription, 101  
material\_index\_address  
    ObjectDescription, 101  
materialIdsBuffer  
    Mesh, 89  
materialIndex  
    ObjLoader, 108  
materials  
    ObjLoader, 109  
materialsBuffer  
    Mesh, 89  
MAX\_FRAME\_DRAWS  
    Globals.hpp, 360  
MAX\_OBJECTS  
    Globals.hpp, 360  
MAX\_TEXTURE\_COUNT  
    host\_device\_shared\_vars.hpp, 411  
maxComputeWorkGroupCount  
    PathTracing, 121  
maxComputeWorkGroupInvocations  
    PathTracing, 122  
maxComputeWorkGroupSize  
    PathTracing, 122  
MemoryHelper.hpp  
    align\_up, 361  
    find\_memory\_type\_index, 361  
Mesh, 76  
    ~Mesh, 80  
    cleanUp, 80  
    createIndexBuffer, 81  
    createMaterialBuffer, 82  
    createMaterialIDBuffer, 83  
    createVertexBuffer, 84  
    device, 88  
    getIndexBuffer, 85

getIndexCount, 86  
 getMaterialIDBuffer, 86  
 getModel, 86  
 getObjectDescription, 87  
 getVertexBuffer, 87  
 getVertexCount, 88  
 index\_count, 88  
 indexBuffer, 89  
 materialIdsBuffer, 89  
 materialsBuffer, 89  
 Mesh, 78, 80  
 model, 89  
 object\_description, 89  
 objectDescriptionBuffer, 90  
 setModel, 88  
 vertex\_count, 90  
 vertexBuffer, 90  
 vulkanBufferManager, 90  
 mesh  
     Model, 99  
 mesh\_model\_index  
     Model, 99  
 messageCallback  
     debug, 11  
 mip\_levels  
     Texture, 218  
 miss\_region  
     Raytracing, 181  
 missShaderBindingTableBuffer  
     Raytracing, 181  
 Model, 91  
     ~Model, 93  
     add\_new\_mesh, 93  
     addSampler, 93  
     addTexture, 94  
     cleanUp, 95  
     device, 99  
     getCustomInstanceIndex, 95  
     getMesh, 96  
     getMeshCount, 96  
     getModel, 96  
     getObjectDescription, 96  
     getPrimitiveCount, 97  
     getTextureCount, 97  
     getTextureList, 98  
     getTextures, 98  
     getTextureSamplers, 98  
     mesh, 99  
     mesh\_model\_index, 99  
     Model, 92  
     model, 99  
     modelTextures, 99  
     modelTextureSamplers, 99  
     set\_model, 98  
     texture\_list, 100  
 model  
     Mesh, 89  
     Model, 99  
     PushConstantRasterizer, 146  
 model\_list  
     Scene, 196  
 modelTextures  
     Model, 99  
 modelTextureSamplers  
     Model, 99  
 mouse\_button\_callback  
     Window, 344  
 mouse\_callback  
     Window, 345  
 mouse\_control  
     Camera, 48  
 mouse\_first\_moved  
     Window, 348  
 movement\_speed  
     Camera, 51  
 near\_plane  
     Camera, 52  
 normal  
     Vertex, 223  
 object\_description  
     Mesh, 89  
 object\_descriptions  
     Scene, 196  
 ObjectDescription, 100  
     index\_address, 101  
     material\_address, 101  
     material\_index\_address, 101  
     vertex\_address, 101  
 objectDescriptionBuffer  
     Mesh, 90  
     VulkanRenderer, 319  
 objectToVkGeometryKHR  
     ASManager, 34  
 ObjLoader, 102  
     command\_pool, 108  
     device, 108  
     indices, 108  
     loadModel, 103  
     loadTexturesAndMaterials, 105  
     loadVertices, 106  
     materialIndex, 108  
     materials, 109  
     ObjLoader, 103  
     textures, 109  
     transfer\_queue, 109  
     vertices, 109  
 ObjMaterial, 110  
     ambient, 110  
     diffuse, 111  
     dissolve, 111  
     emission, 111  
     illum, 111  
     ior, 111  
     shininess, 111  
     specular, 112

textureID, 112  
transmittance, 112  
**ObjMaterial.hpp**  
mat4, 389  
uint, 389  
vec2, 389  
vec3, 389  
vec4, 389  
offscreenTextures  
    Rasterizer, 167  
offscreenTextureSampler  
    PostStage, 142  
onErrorCallback  
    Window.cpp, 501  
operator()  
    std::hash< Vertex >, 76  
operator==  
    Vertex, 222  
  
**PathTracing**, 112  
    ~PathTracing, 114  
    cleanUp, 114  
    computeLimits, 121  
    createPipeline, 114  
    createQueryPool, 116  
    device, 121  
    init, 117  
    maxComputeWorkGroupCount, 121  
    maxComputeWorkGroupInvocations, 122  
    maxComputeWorkGroupSize, 122  
    PathTracing, 113  
    pathTracingTiming, 122  
    pc\_range, 122  
    pipeline, 122  
    pipeline\_layout, 123  
    push\_constant, 123  
    query\_count, 123  
    queryPool, 123  
    queryResults, 123  
    recordCommands, 118  
    shaderHotReload, 120  
    specializationData, 124  
    timeStampPeriod, 124  
**pathTracing**  
    GUIRendererSharedVars, 73  
    VulkanRenderer, 319  
**PathTracing::SpecializationData**, 203  
    specWorkGroupSizeX, 203  
    specWorkGroupSizeY, 203  
    specWorkGroupSizeZ, 204  
**pathTracingTiming**  
    PathTracing, 122  
**pc**  
    Raytracing, 181  
**pc\_range**  
    PathTracing, 122  
**pc\_ranges**  
    Raytracing, 182  
**physical\_device**  
    VulkanDevice, 254  
**pipeline**  
    PathTracing, 122  
**pipeline\_layout**  
    PathTracing, 123  
    PostStage, 142  
    Rasterizer, 167  
    Raytracing, 182  
**pipelineStageForLayout**  
    VulkanImage, 261  
**pitch**  
    Camera, 52  
**pos**  
    Vertex, 223  
**position**  
    Camera, 52  
**post\_descriptor\_pool**  
    VulkanRenderer, 319  
**post\_descriptor\_set**  
    VulkanRenderer, 319  
**post\_descriptor\_set\_layout**  
    VulkanRenderer, 320  
**PostStage**, 124  
    ~PostStage, 126  
    cleanUp, 126  
    createDepthbufferImage, 127  
    createFramebuffer, 128  
    createGraphicsPipeline, 129  
    createOffscreenTextureSampler, 133  
    createPushConstantRange, 134  
    createRenderpass, 134  
    depth\_format, 141  
    depthBufferImage, 141  
    device, 141  
    framebuffers, 141  
    getOffscreenSampler, 136  
    getRenderPass, 137  
    graphics\_pipeline, 141  
    init, 137  
    offscreenTextureSampler, 142  
    pipeline\_layout, 142  
    PostStage, 126  
    push\_constant\_range, 142  
    recordCommands, 138  
    render\_pass, 142  
    shaderHotReload, 140  
    vulkanSwapChain, 142  
**postStage**  
    VulkanRenderer, 320  
**presentation\_family**  
    QueueFamilyIndices, 149  
**presentation\_mode**  
    SwapChainDetails, 205  
**presentation\_queue**  
    VulkanDevice, 254  
**projection**  
    GlobalUBO, 61  
**push\_constant**

PathTracing, 123  
 push\_constant\_range  
     PostStage, 142  
     Rasterizer, 168  
 pushConstant  
     Rasterizer, 168  
 PushConstantPathTracing, 143  
     clearColor, 144  
     height, 144  
     width, 144  
 PushConstantPathTracing.hpp  
     mat4, 371  
     uint, 371  
     vec2, 371  
     vec3, 371  
     vec4, 372  
 PushConstantPost, 145  
     aspect\_ratio, 145  
 PushConstantPost.hpp  
     mat4, 373  
     uint, 373  
     vec2, 373  
     vec3, 373  
     vec4, 373  
 PushConstantRasterizer, 146  
     model, 146  
 PushConstantRasterizer.hpp  
     mat4, 374  
     uint, 374  
     vec2, 375  
     vec3, 375  
     vec4, 375  
 PushConstantRaytracing, 147  
     clear\_color, 147  
 PushConstantRayTracing.hpp  
     mat4, 376  
     uint, 376  
     vec2, 376  
     vec3, 376  
     vec4, 377  
 query\_count  
     PathTracing, 123  
 queryPool  
     PathTracing, 123  
 queryResults  
     PathTracing, 123  
 QueueFamilyIndices, 148  
     compute\_family, 149  
     graphics\_family, 149  
     is\_valid, 148  
     presentation\_family, 149  
 range\_info  
     BuildAccelerationStructure, 42  
 Rasterizer, 150  
     ~Rasterizer, 151  
     cleanUp, 152  
     commandBufferManager, 166  
     createFramebuffer, 152  
     createGraphicsPipeline, 153  
     createPushConstantRange, 157  
     createRenderPass, 157  
     createTextures, 159  
     depthBufferImage, 166  
     device, 167  
     framebuffer, 167  
     getOffscreenTexture, 161  
     graphics\_pipeline, 167  
     init, 162  
     offscreenTextures, 167  
     pipeline\_layout, 167  
     push\_constant\_range, 168  
     pushConstant, 168  
     Rasterizer, 151  
     recordCommands, 163  
     render\_pass, 168  
     setPushConstant, 165  
     shaderHotReload, 165  
     vulkanSwapChain, 168  
 rasterizer  
     VulkanRenderer, 320  
 raygenShaderBindingTableBuffer  
     Raytracing, 182  
 Raytracing, 169  
     ~Raytracing, 170  
     call\_region, 180  
     cleanUp, 170  
     createGraphicsPipeline, 171  
     createPCRange, 174  
     createSBT, 175  
     device, 180  
     graphicsPipeline, 180  
     hit\_region, 181  
     hitShaderBindingTableBuffer, 181  
     init, 176  
     miss\_region, 181  
     missShaderBindingTableBuffer, 181  
     pc, 181  
     pc\_ranges, 182  
     pipeline\_layout, 182  
     raygenShaderBindingTableBuffer, 182  
     Raytracing, 170  
     raytracing\_properties, 182  
     recordCommands, 177  
     rgen\_region, 182  
     shader\_groups, 183  
     shaderBindingTableBuffer, 183  
     shaderHotReload, 179  
     vulkanSwapChain, 183  
 raytracing  
     GUIRendererSharedVars, 73  
 raytracing\_properties  
     Raytracing, 182  
 raytracingDescriptorPool  
     VulkanRenderer, 320  
 raytracingDescriptorSet

VulkanRenderer, 320  
raytracingDescriptorSetLayout  
    VulkanRenderer, 321  
raytracingStage  
    VulkanRenderer, 321  
read  
    File, 58  
readCharSequence  
    File, 59  
record\_commands  
    VulkanRenderer, 304  
recordCommands  
    PathTracing, 118  
    PostStage, 138  
    Rasterizer, 163  
    Raytracing, 177  
render  
    GUI, 69  
render\_finished  
    VulkanRenderer, 321  
render\_pass  
    PostStage, 142  
    Rasterizer, 168  
reset\_framebuffer\_has\_changed  
    Window, 346  
rgen\_region  
    Raytracing, 182  
right  
    Camera, 52  
run  
    App, 21  
  
Scene, 184  
    ~Scene, 185  
    add\_model, 186  
    add\_object\_description, 186  
    cleanUp, 186  
    get\_model\_list, 186  
    getGuiSceneSharedVars, 187  
    getIndexBuffer, 187  
    getIndexCount, 188  
    getMeshCount, 188  
    getModelCount, 189  
    getModelMatrix, 189  
    getNumberMeshes, 190  
    getNumberObjectDescriptions, 190  
    getObjectDescriptions, 190  
    getTextureCount, 191  
    getTextures, 191  
    getTextureSampler, 192  
    getVertexBuffer, 192  
    guiSceneSharedVars, 195  
    loadModel, 193  
    model\_list, 196  
    object\_descriptions, 196  
    Scene, 185  
    update\_model\_matrix, 194  
    update\_user\_input, 195  
scene  
    VulkanRenderer, 321  
sceneConfig, 13  
    getModelFile, 13  
    getModelMatrix, 14  
SceneUBO, 196  
    cam\_pos, 197  
    light\_dir, 197  
    view\_dir, 198  
sceneUBO  
    VulkanRenderer, 321  
SceneUBO.hpp  
    mat4, 380  
    uint, 380  
    vec2, 380  
    vec3, 380  
    vec4, 380  
sceneUBOBuffer  
    VulkanRenderer, 322  
set  
    CMakeLists.txt, 504–506  
set\_buffer\_size  
    Window, 346  
set\_camera\_position  
    Camera, 49  
set\_far\_plane  
    Camera, 49  
set\_fov  
    Camera, 50  
set\_model  
    Model, 98  
set\_near\_plane  
    Camera, 50  
setImage  
    Texture, 216  
    VulkanImage, 261  
setImageView  
    Texture, 217  
    VulkanImageView, 269  
setModel  
    Mesh, 88  
setPushConstant  
    Rasterizer, 165  
setupDebugging  
    debug, 11  
shader\_groups  
    Raytracing, 183  
shader\_hot\_reload\_triggered  
    GUIRendererSharedVars, 73  
shaderBindingTableBuffer  
    Raytracing, 183  
ShaderHelper, 198  
    ~ShaderHelper, 199  
    compileShader, 199  
    createShaderModule, 200  
    getShaderSpvDir, 201  
    ShaderHelper, 199  
    target, 202  
shaderHotReload

PathTracing, 120  
 PostStage, 140  
 Rasterizer, 165  
 Raytracing, 179  
 VulkanRenderer, 305  
 sharedRenderDescriptorSet  
     VulkanRenderer, 322  
 sharedRenderDescriptorsetLayout  
     VulkanRenderer, 322  
 shininess  
     ObjMaterial, 111  
 single\_bla  
     BuildAccelerationStructure, 42  
 size\_info  
     BuildAccelerationStructure, 42  
 specializationData  
     PathTracing, 124  
 specular  
     ObjMaterial, 112  
 specWorkGroupSizeX  
     PathTracing::SpecializationData, 203  
 specWorkGroupSizeY  
     PathTracing::SpecializationData, 203  
 specWorkGroupSizeZ  
     PathTracing::SpecializationData, 204  
 std, 15  
 std::hash< Vertex >, 75  
     operator(), 76  
 surface  
     VulkanDevice, 255  
     VulkanRenderer, 322  
 surface\_capabilities  
     SwapChainDetails, 205  
 swap\_chain\_extent  
     VulkanSwapChain, 333  
 swap\_chain\_image\_format  
     VulkanSwapChain, 334  
 swap\_chain\_images  
     VulkanSwapChain, 334  
 swapchain  
     VulkanSwapChain, 334  
 SwapChainDetails, 204  
     formats, 205  
     presentation\_mode, 205  
     surface\_capabilities, 205  
 target  
     ShaderHelper, 202  
 TEST  
     commitSuite.cpp, 506  
     compileSuite.cpp, 508  
 Texture, 206  
     ~Texture, 207  
     cleanUp, 208  
     commandBufferManager, 218  
     createFromFile, 208  
     createImage, 210  
     createImageView, 211  
     generateMipMaps, 212  
     getImage, 214  
     getImageView, 214  
     getMipLevel, 215  
     getVulkanImage, 215  
     getVulkanImageView, 215  
     loadTextureData, 216  
     mip\_levels, 218  
     setImage, 216  
     setImageView, 217  
     Texture, 207  
     vulkanBufferManager, 218  
     vulkanImage, 218  
     vulkanImageView, 218  
 texture\_coords  
     Vertex, 224  
 texture\_list  
     Model, 100  
 textureID  
     ObjMaterial, 112  
 textures  
     ObjLoader, 109  
 timeStampPeriod  
     PathTracing, 124  
 tlas  
     ASManager, 36  
 TopLevelAccelerationStructure, 219  
     vulkanAS, 220  
     vulkanBuffer, 220  
 transfer\_queue  
     ObjLoader, 109  
 transitionImageLayout  
     VulkanImage, 262, 263  
 transmittance  
     ObjMaterial, 112  
 turn\_speed  
     Camera, 52  
 uint  
     GlobalUBO.hpp, 367  
     ObjMaterial.hpp, 389  
     PushConstantPathTracing.hpp, 371  
     PushConstantPost.hpp, 373  
     PushConstantRasterizer.hpp, 374  
     PushConstantRayTracing.hpp, 376  
     SceneUBO.hpp, 380  
 up  
     Camera, 53  
 update  
     Camera, 50  
 update\_model\_matrix  
     Scene, 194  
 update\_raytracing\_descriptor\_set  
     VulkanRenderer, 307  
 update\_uniform\_buffers  
     VulkanRenderer, 308  
 update\_user\_input  
     Scene, 195  
 update\_viewport  
     Window, 346

updatePostDescriptorSets  
    VulkanRenderer, 309  
updateRaytracingDescriptorSets  
    VulkanRenderer, 310  
updateStateDueToUserInput  
    VulkanRenderer, 312  
updateTexturesInSharedRenderDescriptorSet  
    VulkanRenderer, 312  
updateUniforms  
    VulkanRenderer, 314  
Utilities.hpp  
    ENABLE\_VALIDATION\_LAYERS, 363

validationLayerCount  
    debug, 12  
validationLayerNames  
    debug, 13  
validationLayers  
    VulkanInstance, 276

vec2  
    GlobalUBO.hpp, 367  
    ObjMaterial.hpp, 389  
    PushConstantPathTracing.hpp, 371  
    PushConstantPost.hpp, 373  
    PushConstantRasterizer.hpp, 375  
    PushConstantRayTracing.hpp, 376  
    SceneUBO.hpp, 380

vec3  
    GlobalUBO.hpp, 368  
    ObjMaterial.hpp, 389  
    PushConstantPathTracing.hpp, 371  
    PushConstantPost.hpp, 373  
    PushConstantRasterizer.hpp, 375  
    PushConstantRayTracing.hpp, 376  
    SceneUBO.hpp, 380

vec4  
    GlobalUBO.hpp, 368  
    ObjMaterial.hpp, 389  
    PushConstantPathTracing.hpp, 372  
    PushConstantPost.hpp, 373  
    PushConstantRasterizer.hpp, 375  
    PushConstantRayTracing.hpp, 377  
    SceneUBO.hpp, 380

Vertex, 221  
    color, 223  
    normal, 223  
    operator==, 222  
    pos, 223  
    texture\_coords, 224  
    Vertex, 222

vertex, 15  
    getVertexInputAttributeDesc, 15

vertex\_address  
    ObjectDescription, 101

vertex\_count  
    Mesh, 90

vertexBuffer  
    Mesh, 90

vertices  
    ObjLoader, 109  
view  
    GlobalUBO, 61  
view\_dir  
    SceneUBO, 198

vkCreateDebugUtilsMessengerEXT  
    debug, 13

vkDestroyDebugUtilsMessengerEXT  
    debug, 13

vmaAllocator  
    Allocator, 19

vulkanAS  
    BottomLevelAccelerationStructure, 40  
    TopLevelAccelerationStructure, 220

VulkanBuffer, 224  
    ~VulkanBuffer, 226  
    buffer, 230  
    bufferMemory, 230  
    cleanUp, 226  
    create, 227  
    created, 230  
    device, 231  
    getBuffer, 229  
    getBufferMemory, 229  
    VulkanBuffer, 226

vulkanBuffer  
    BottomLevelAccelerationStructure, 40  
    TopLevelAccelerationStructure, 220

VulkanBufferManager, 231  
    ~VulkanBufferManager, 233  
    commandBufferManager, 237  
    copyBuffer, 233  
    copyImageBuffer, 235  
    createBufferAndUploadVectorOnDevice, 236  
    VulkanBufferManager, 233

vulkanBufferManager  
    ASManager, 37  
    Mesh, 90  
    Texture, 218  
    VulkanRenderer, 322

VulkanDevice, 238  
    ~VulkanDevice, 240  
    check\_device\_extension\_support, 240  
    check\_device\_suitable, 241  
    cleanUp, 242  
    compute\_queue, 253  
    create\_logical\_device, 242  
    device\_extensions, 253  
    device\_extensions\_for\_raytracing, 253  
    device\_properties, 253  
    get\_physical\_device, 245  
    getComputeQueue, 246  
    getGraphicsQueue, 246  
    getLogicalDevice, 246  
    getPhysicalDevice, 248  
    getPhysicalDeviceProperties, 249  
    getPresentationQueue, 249  
    getQueueFamilies, 249, 250

getSwapchainDetails, 251, 252  
 graphics\_queue, 254  
 instance, 254  
 logical\_device, 254  
 physical\_device, 254  
 presentation\_queue, 254  
 surface, 255  
 VulkanDevice, 239  
 vulkanDevice  
     ASManager, 37  
 VulkanImage, 255  
     ~VulkanImage, 257  
     accessFlagsForImageLayout, 257  
     cleanUp, 258  
     commandBufferManager, 265  
     create, 259  
     device, 265  
     getImage, 260  
     image, 265  
     imageMemory, 265  
     pipelineStageForLayout, 261  
     setImage, 261  
     transitionImageLayout, 262, 263  
     VulkanImage, 257  
 vulkanImage  
     Texture, 218  
 VulkanImageView, 266  
     ~VulkanImageView, 267  
     cleanUp, 267  
     create, 268  
     device, 270  
     getImageView, 269  
     imageView, 270  
     setImageView, 269  
     VulkanImageView, 267  
 vulkanImageView  
     Texture, 218  
 VulkanInstance, 270  
     ~VulkanInstance, 273  
     check\_instance\_extension\_support, 273  
     check\_validation\_layer\_support, 274  
     cleanUp, 274  
     getVulkanInstance, 275  
     instance, 276  
     validationLayers, 276  
     VulkanInstance, 272  
 VulkanRenderer, 276  
     ~VulkanRenderer, 280  
     allocator, 315  
     asManager, 316  
     checkChangedFramebufferSize, 281  
     cleanUp, 283  
     cleanUpCommandPools, 284  
     cleanUpSync, 284  
     cleanUpUBOs, 285  
     command\_buffers, 316  
     commandBufferManager, 316  
     compute\_command\_pool, 316  
     create\_command\_buffers, 285  
     create\_command\_pool, 286  
     create\_object\_description\_buffer, 287  
     create\_post\_descriptor\_layout, 289  
     create\_surface, 291  
     create\_uniform\_buffers, 291  
     createDescriptorPoolSharedRenderStages, 292  
     createRaytracingDescriptorPool, 294  
     createRaytracingDescriptorSetLayouts, 295  
     createRaytracingDescriptorSets, 295  
     createSharedRenderDescriptorSet, 296  
     createSharedRenderDescriptorSetLayouts, 298  
     createSynchronization, 299  
     current\_frame, 316  
     descriptorPoolSharedRenderStages, 317  
     device, 317  
     drawFrame, 300  
     finishAllRenderCommands, 303  
     globalUBO, 317  
     globalUBOBuffer, 317  
     graphics\_command\_pool, 318  
     gui, 318  
     image\_available, 318  
     images\_in\_flight\_fences, 318  
     in\_flight\_fences, 318  
     instance, 319  
     objectDescriptionBuffer, 319  
     pathTracing, 319  
     post\_descriptor\_pool, 319  
     post\_descriptor\_set, 319  
     post\_descriptor\_set\_layout, 320  
     postStage, 320  
     rasterizer, 320  
     raytracingDescriptorPool, 320  
     raytracingDescriptorSet, 320  
     raytracingDescriptorsetLayout, 321  
     raytracingStage, 321  
     record\_commands, 304  
     render\_finished, 321  
     scene, 321  
     sceneUBO, 321  
     sceneUBOBuffer, 322  
     shaderHotReload, 305  
     sharedRenderDescriptorSet, 322  
     sharedRenderDescriptorsetLayout, 322  
     surface, 322  
     update\_raytracing\_descriptor\_set, 307  
     update\_uniform\_buffers, 308  
     updatePostDescriptorSets, 309  
     updateRaytracingDescriptorSets, 310  
     updateStateDueToUserInput, 312  
     updateTexturesInSharedRenderDescriptorSet, 312  
     updateUniforms, 314  
     vulkanBufferManager, 322  
     VulkanRenderer, 278  
     vulkanSwapChain, 323  
     window, 323  
     VulkanSwapChain, 324

~VulkanSwapChain, 325  
choose\_best\_presentation\_mode, 325  
choose\_best\_surface\_format, 326  
choose\_swap\_extent, 326  
cleanUp, 327  
device, 333  
getNumberSwapChainImages, 328  
getSwapChain, 329  
getSwapChainExtent, 329  
getSwapChainFormat, 330  
getSwapChainImage, 330  
initVulkanContext, 331  
swap\_chain\_extent, 333  
swap\_chain\_image\_format, 334  
swap\_chain\_images, 334  
swapchain, 334  
VulkanSwapChain, 325  
window, 334  
vulkanSwapChain  
    PostStage, 142  
    Rasterizer, 168  
    Raytracing, 183  
    VulkanRenderer, 323  
width  
    PushConstantPathTracing, 144  
Window, 335  
    ~Window, 338  
    cleanUp, 338  
    framebuffer\_resized, 347  
    framebuffer\_size\_callback, 338  
    framebuffer\_size\_has\_changed, 339  
    get\_buffer\_height, 339  
    get\_buffer\_width, 339  
    get\_height, 340  
    get\_keys, 340  
    get\_should\_close, 340  
    get\_width, 340  
    get\_window, 341  
    get\_x\_change, 341  
    get\_y\_change, 341  
    init\_callbacks, 342  
    initialize, 342  
    key\_callback, 343  
    keys, 347  
    last\_x, 347  
    last\_y, 347  
    main\_window, 348  
    mouse\_button\_callback, 344  
    mouse\_callback, 345  
    mouse\_first\_moved, 348  
    reset\_framebuffer\_has\_changed, 346  
    set\_buffer\_size, 346  
    update\_viewport, 346  
Window, 336, 337  
    window\_buffer\_height, 348  
    window\_buffer\_width, 348  
    window\_height, 348  
    window\_width, 349  
    x\_change, 349  
    y\_change, 349  
window  
    GUI, 72  
    VulkanRenderer, 323  
    VulkanSwapChain, 334  
Window.cpp  
    onErrorCallback, 501  
window\_buffer\_height  
    Window, 348  
window\_buffer\_width  
    Window, 348  
window\_height  
    Window, 348  
window\_width  
    Window, 349  
world\_up  
    Camera, 53  
x\_change  
    Window, 349  
y\_change  
    Window, 349  
yaw  
    Camera, 53