GraphicsEngine

Generated by Doxygen 1.9.4

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Namespace Documentation

3.1 debug Namespace Reference

Functions

- VKAPI_ATTR VkBool32 VKAPI_CALL debugUtilsMessengerCallback (VkDebugUtilsMessageSeverity↩ FlagBitsEXT messageSeverity, VkDebugUtilsMessageTypeFlagsEXT messageType, const VkDebugUtils↩ MessengerCallbackDataEXT ∗pCallbackData, void ∗pUserData)
- void setupDebugging (VkInstance instance, VkDebugReportFlagsEXT flags, VkDebugReportCallbackEXT callBack)
- void freeDebugCallback (VkInstance instance)

Variables

- PFN_vkCreateDebugUtilsMessengerEXT vkCreateDebugUtilsMessengerEXT
- PFN_vkDestroyDebugUtilsMessengerEXT vkDestroyDebugUtilsMessengerEXT
- VkDebugUtilsMessengerEXT debugUtilsMessenger

3.1.1 Function Documentation

3.1.1.1 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:

3.1.1.2 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.

3.1.1.3 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.

Here is the call graph for this function:

3.1.2 Variable Documentation

3.1.2.1 debugUtilsMessenger

```
VkDebugUtilsMessengerEXT debug::debugUtilsMessenger
```

Definition at line 8 of file VulkanDebug.cpp.

Referenced by freeDebugCallback(), and setupDebugging().

3.1.2.2 vkCreateDebugUtilsMessengerEXT

```
PFN_vkCreateDebugUtilsMessengerEXT debug::vkCreateDebugUtilsMessengerEXT
```

Definition at line 6 of file VulkanDebug.cpp.

Referenced by setupDebugging().

3.1.2.3 vkDestroyDebugUtilsMessengerEXT

```
PFN_vkDestroyDebugUtilsMessengerEXT debug::vkDestroyDebugUtilsMessengerEXT
```

Definition at line 7 of file VulkanDebug.cpp.

Referenced by freeDebugCallback(), and setupDebugging().

3.2 sceneConfig Namespace Reference

Functions

- std::string getModelFile ()
- glm::mat4 getModelMatrix ()

3.2.1 Function Documentation

3.2.1.1 getModelFile()

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

Definition at line 7 of file SceneConfig.cpp.

```
00007                                {
00008   std::stringstream modelFile;
00009   modelFile « CMAKELISTS_DIR;
00010 #if NDEBUG
00011   modelFile « "/Resources/Model/crytek-sponza/";
00012   modelFile « "sponza_triag.obj";
00013
00014 #else
00015 #ifdef SULO_MODE
00016   modelFile « "/Resources/Model/Sulo/";
00017   modelFile « "SuloLongDongLampe_v2.obj";
00018 #else
00019   modelFile « "/Resources/Model/VikingRoom/";
00020   modelFile « "viking_room.obj";
00021 #endif
00022 #endif
00023
00024   return modelFile.str();
00025   // std::string modelFile =
00026   // "../Resources/Model/crytek-sponza/sponza_triag.obj"; std::string modelFile
00027   // = "../Resources/Model/Dinosaurs/dinosaurs.obj"; std::string modelFile =
00028   // "../Resources/Model/Pillum/PilumPainting_export.obj"; std::string modelFile
00029   // = "../Resources/Model/sibenik/sibenik.obj"; std::string modelFile =
00030   // "../Resources/Model/sportsCar/sportsCar.obj"; std::string modelFile =
00031   // "../Resources/Model/StanfordDragon/dragon.obj"; std::string modelFile =
00032   // "../Resources/Model/CornellBox/CornellBox-Sphere.obj"; std::string
00033   // modelFile = "../Resources/Model/bunny/bunny.obj"; std::string modelFile =
00034   // "../Resources/Model/buddha/buddha.obj"; std::string modelFile =
00035   // "../Resources/Model/bmw/bmw.obj"; std::string modelFile =
00036   // "../Resources/Model/testScene.obj"; std::string modelFile =
00037   // "../Resources/Model/San_Miguel/san-miguel-low-poly.obj";
00038 }
```

3.2.1.2 getModelMatrix()

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

Definition at line 40 of file SceneConfig.cpp.

```
00040                                {
00041   glm::mat4 modelMatrix(1.0f);
00042
00043 #if NDEBUG
00044
00045   // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00046   // -50.0f));
00047   modelMatrix = glm::scale(modelMatrix, glm::vec3(1.0f, 1.0f, 1.0f));
00048   /*dragon_model = glm::rotate(dragon_model, glm::radians(-90.f),
00049     glm::vec3(1.0f, 0.0f, 0.0f)); dragon_model = glm::rotate(dragon_model,
```

```
00050       glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));*/
00051
00052 #else
00053
00054 // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00055 // -50.0f));
00056 #if SULO_MODE
00057   modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00058 #else
00059   modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00060   modelMatrix = glm::rotate(modelMatrix, glm::radians(-90.f),
00061                             glm::vec3(1.0f, 0.0f, 0.0f));
00062   modelMatrix =
00063       glm::rotate(modelMatrix, glm::radians(90.f), glm::vec3(0.0f, 0.0f, 1.0f));
00064 #endif
00065
00066 #endif
00067
00068   return modelMatrix;
00069 }
```

3.3 vertex Namespace Reference

Functions

- std::array< VkVertexInputAttributeDescription, 4 > getVertexInputAttributeDesc ()

3.3.1 Function Documentation

3.3.1.1 getVertexInputAttributeDesc()

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

Definition at line 20 of file Vertex.cpp.

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

Chapter 4

File Documentation

4.1 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/App.cpp File Reference

```
#include "App.h"
#include <vulkan/vulkan.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/mat4x4.hpp>
#include <iostream>
#include <memory>
#include <stdexcept>
#include <vector>
#include "GUI.h"
#include "VulkanRenderer.hpp"
#include "Window.h"
```

Include dependency graph for App.cpp:

Macros

- #define GLFW_INCLUDE_NONE
- #define GLFW_INCLUDE_VULKAN
- #define GLM_FORCE_RADIANS
- #define GLM_FORCE_DEPTH_ZERO_TO_ONE

4.1.1 Macro Definition Documentation

4.1.1.1 GLFW_INCLUDE_NONE

```
#define GLFW_INCLUDE_NONE
```

Definition at line 4 of file App.cpp.

4.1.1.2 GLFW_INCLUDE_VULKAN

```
#define GLFW_INCLUDE_VULKAN
```

Definition at line 5 of file App.cpp.

4.1.1.3 GLM_FORCE_DEPTH_ZERO_TO_ONE

```
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
```

Definition at line 9 of file App.cpp.

4.1.1.4 GLM_FORCE_RADIANS

```
#define GLM_FORCE_RADIANS
```

Definition at line 8 of file App.cpp.

4.2 App.cpp

Go to the documentation of this file.

```
00001 #include "App.h"
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.h"
00019 #include "VulkanRenderer.hpp"
00020 #include "Window.h"
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() {}
```

4.3 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/gui/GUI.cpp File Reference

```
#include "GUI.h"
#include "QueueFamilyIndices.h"
#include "Utilities.h"
#include "VulkanDevice.h"
```

Include dependency graph for GUI.cpp:

4.4 GUI.cpp

Go to the documentation of this file.

```
00001 #include "GUI.h"
00002
00003 #include "QueueFamilyIndices.h"
00004 #include "Utilities.h"
00005 #include "VulkanDevice.h"
00006
00007 GUI::GUI(Window* window) { this->window = window; }
00008
00009 void GUI::initializeVulkanContext(VulkanDevice* device,
00010                                   const VkInstance& instance,
00011                                   const VkRenderPass& post_render_pass,
00012                                   const VkCommandPool& graphics_command_pool) {
00013   this->device = device;
00014
00015   create_gui_context(window, instance, post_render_pass);
00016   create_fonts_and_upload(graphics_command_pool);
00017 }
00018
00019 void GUI::render() {
00020   // Start the Dear ImGui frame
00021   ImGui_ImplVulkan_NewFrame();
00022   ImGui_ImplGlfw_NewFrame();
00023   ImGui::NewFrame();
00024
00025   // ImGui::ShowDemoWindow();
00026
```

```
00027   // render your GUI
00028   ImGui::Begin("GUI v1.4.4");
00029
00030   if (ImGui::CollapsingHeader("Hot shader reload")) {
00031     if (ImGui::Button("All shader!")) {
00032       guiRendererSharedVars.shader_hot_reload_triggered = true;
00033     }
00034   }
00035
00036   ImGui::Separator();
00037
00038   static int e = 0;
00039   ImGui::RadioButton("Rasterizer", &e, 0);
00040   ImGui::SameLine();
00041   ImGui::RadioButton("Raytracing", &e, 1);
00042   ImGui::SameLine();
00043   ImGui::RadioButton("Path tracing", &e, 2);
00044
00045   switch (e) {
00046     case 0:
00047       guiRendererSharedVars.raytracing = false;
00048       guiRendererSharedVars.pathTracing = false;
00049       break;
00050     case 1:
00051       guiRendererSharedVars.raytracing = true;
00052       guiRendererSharedVars.pathTracing = false;
00053       break;
00054     case 2:
00055       guiRendererSharedVars.raytracing = false;
00056       guiRendererSharedVars.pathTracing = true;
00057       break;
00058   }
00059   // ImGui::Checkbox("Ray tracing", &guiRendererSharedVars.raytracing);
00060
00061   ImGui::Separator();
00062
00063   if (ImGui::CollapsingHeader("Graphic Settings")) {
00064     if (ImGui::TreeNode("Directional Light")) {
00065       ImGui::Separator();
00066       ImGui::SliderFloat("Ambient intensity",
00067                          &guiSceneSharedVars.direcional_light_radiance, 0.0f,
00068                          50.0f);
00069       ImGui::Separator();
00070       // Edit a color (stored as ~4 floats)
00071       ImGui::ColorEdit3("Directional Light Color",
00072                         guiSceneSharedVars.directional_light_color);
00073       ImGui::Separator();
00074       ImGui::SliderFloat3("Light Direction",
00075                           guiSceneSharedVars.directional_light_direction, -1.f,
00076                           1.0f);
00077
00078       ImGui::TreePop();
00079     }
00080   }
00081
00082   ImGui::Separator();
00083
00084   if (ImGui::CollapsingHeader("GUI Settings")) {
00085     ImGuiStyle& style = ImGui::GetStyle();
00086
00087     if (ImGui::SliderFloat("Frame Rounding", &style.FrameRounding, 0.0f, 12.0f,
00088                            "%.0f")) {
00089       style.GrabRounding = style.FrameRounding;  // Make GrabRounding always the
00090                                                  // same value as FrameRounding
00091     }
00092     {
00093       bool border = (style.FrameBorderSize > 0.0f);
00094       if (ImGui::Checkbox("FrameBorder", &border)) {
00095         style.FrameBorderSize = border ? 1.0f : 0.0f;
00096       }
00097     }
00098     ImGui::SliderFloat("WindowRounding", &style.WindowRounding, 0.0f, 12.0f,
00099                        "%.0f");
00100   }
00101
00102   ImGui::Separator();
00103
00104   if (ImGui::CollapsingHeader("KEY Bindings")) {
00105     ImGui::Text(
00106         "WASD for moving Forward, backward and to the side\n QE for rotating ");
00107   }
00108
00109   ImGui::Separator();
00110
00111   ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
00112               1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
00113
```

```
00114   ImGui::End();
00115 }
00116
00117 void GUI::cleanUp() {
00118   // clean up of GUI stuff
00119   ImGui_ImplVulkan_Shutdown();
00120   ImGui_ImplGlfw_Shutdown();
00121   ImGui::DestroyContext();
00122   vkDestroyDescriptorPool(device->getLogicalDevice(), gui_descriptor_pool,
00123                           nullptr);
00124 }
00125
00126 void GUI::create_gui_context(Window* window, const VkInstance& instance,
00127                              const VkRenderPass& post_render_pass) {
00128   IMGUI_CHECKVERSION();
00129   ImGui::CreateContext();
00130   ImGuiIO& io = ImGui::GetIO();
00131   (void)io;
00132
00133   float size_pixels = 18;
00134
00135   std::stringstream fontDir;
00136   fontDir « CMAKELISTS_DIR;
00137   fontDir « "/ExternalLib/IMGUI/misc/fonts/";
00138
00139   std::stringstream robo_font;
00140   robo_font « fontDir.str() « "Roboto-Medium.ttf";
00141   std::stringstream Cousine_font;
00142   Cousine_font « fontDir.str() « "Cousine-Regular.ttf";
00143   std::stringstream DroidSans_font;
00144   DroidSans_font « fontDir.str() « "DroidSans.ttf";
00145   std::stringstream Karla_font;
00146   Karla_font « fontDir.str() « "Karla-Regular.ttf";
00147   std::stringstream proggy_clean_font;
00148   proggy_clean_font « fontDir.str() « "ProggyClean.ttf";
00149   std::stringstream proggy_tiny_font;
00150   proggy_tiny_font « fontDir.str() « "ProggyTiny.ttf";
00151
00152   io.Fonts->AddFontFromFileTTF(robo_font.str().c_str(), size_pixels);
00153   io.Fonts->AddFontFromFileTTF(Cousine_font.str().c_str(), size_pixels);
00154   io.Fonts->AddFontFromFileTTF(DroidSans_font.str().c_str(), size_pixels);
00155   io.Fonts->AddFontFromFileTTF(Karla_font.str().c_str(), size_pixels);
00156   io.Fonts->AddFontFromFileTTF(proggy_clean_font.str().c_str(), size_pixels);
00157   io.Fonts->AddFontFromFileTTF(proggy_tiny_font.str().c_str(), size_pixels);
00158
00159   ImGui::PushStyleVar(ImGuiStyleVar_WindowRounding, 10);
00160   ImGui::PushStyleVar(ImGuiStyleVar_FrameRounding, 10);
00161   ImGui::PushStyleVar(ImGuiStyleVar_FrameBorderSize, 1);
00162   io.ConfigFlags |=
00163       ImGuiConfigFlags_NavEnableKeyboard;  // Enable Keyboard Controls
00164   io.ConfigFlags |= ImGuiConfigFlags_NavEnableSetMousePos;
00165   io.WantCaptureMouse = true;
00166   // io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad;      // Enable Gamepad
00167   // Controls
00168
00169   // Setup Dear ImGui style
00170   ImGui::StyleColorsDark();
00171   // ImGui::StyleColorsClassic();
00172
00173   ImGui_ImplGlfw_InitForVulkan(window->get_window(), false);
00174
00175   // Create Descriptor Pool
00176   VkDescriptorPoolSize gui_pool_sizes[] = {
00177       {VK_DESCRIPTOR_TYPE_SAMPLER, 10},
00178       {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 10},
00179       {VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 10},
00180       {VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 10},
00181       {VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, 10},
00182       {VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, 10},
00183       {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 10},
00184       {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 10},
00185       {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 10},
00186       {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, 10},
00187       {VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, 100}};
00188
00189   VkDescriptorPoolCreateInfo gui_pool_info = {};
00190   gui_pool_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00191   gui_pool_info.flags = VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT;
00192   gui_pool_info.maxSets = 10 * IM_ARRAYSIZE(gui_pool_sizes);
00193   gui_pool_info.poolSizeCount = (uint32_t)IM_ARRAYSIZE(gui_pool_sizes);
00194   gui_pool_info.pPoolSizes = gui_pool_sizes;
00195
00196   VkResult result =
00197       vkCreateDescriptorPool(device->getLogicalDevice(), &gui_pool_info,
00198                              nullptr, &gui_descriptor_pool);
00199   ASSERT_VULKAN(result, "Failed to create a gui descriptor pool!")
00200
```

```
00201   QueueFamilyIndices indices = device->getQueueFamilies();
00202
00203   ImGui_ImplVulkan_InitInfo init_info = {};
00204   init_info.Instance = instance;
00205   init_info.PhysicalDevice = device->getPhysicalDevice();
00206   init_info.Device = device->getLogicalDevice();
00207   init_info.QueueFamily = indices.graphics_family;
00208   init_info.Queue = device->getGraphicsQueue();
00209   init_info.DescriptorPool = gui_descriptor_pool;
00210   init_info.PipelineCache = VK_NULL_HANDLE;
00211   init_info.MinImageCount = 2;
00212   init_info.ImageCount = MAX_FRAME_DRAWS;
00213   init_info.Allocator = VK_NULL_HANDLE;
00214   init_info.CheckVkResultFn = VK_NULL_HANDLE;
00215   init_info.Subpass = 0;
00216   init_info.MSAASamples = VK_SAMPLE_COUNT_1_BIT;
00217
00218   ImGui_ImplVulkan_Init(&init_info, post_render_pass);
00219 }
00220
00221 void GUI::create_fonts_and_upload(const VkCommandPool& graphics_command_pool) {
00222   VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
00223       device->getLogicalDevice(), graphics_command_pool);
00224   ImGui_ImplVulkan_CreateFontsTexture(command_buffer);
00225   commandBufferManager.endAndSubmitCommandBuffer(
00226       device->getLogicalDevice(), graphics_command_pool,
00227       device->getGraphicsQueue(), command_buffer);
00228
00229   // wait until no actions being run on device before destroying
00230   vkDeviceWaitIdle(device->getLogicalDevice());
00231   // clear font textures from cpu data
00232   ImGui_ImplVulkan_DestroyFontUploadObjects();
00233 }
00234
00235 GUI::~GUI() {}
```

4.5 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/Main.cpp File Reference

`#include "App.h"`
Include dependency graph for Main.cpp:

Functions

- int main ()

4.5.1 Function Documentation

4.5.1.1 main()

```
int main ( )
```

Definition at line 3 of file Main.cpp.

```
00003           {
00004   App application;
00005   return application.run();
00006 }
```

4.6 Main.cpp

Go to the documentation of this file.

```
00001 #include "App.h"
00002
00003 int main() {
00004   App application;
00005   return application.run();
00006 }
```

4.7 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/memory/↩ Allocator.cpp File Reference

```
#include "Allocator.h"
#include "Utilities.h"
```

Include dependency graph for Allocator.cpp:

4.8 Allocator.cpp

Go to the documentation of this file.

```
00001 #include "Allocator.h"
00002
00003 #include "Utilities.h"
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() {}
```

4.9 C:/Users/jonas/Desktop/GraphicsEngineVulkan/↩ Src/renderer/accelerationStructures/ASManager.cpp File Reference

```
#include "ASManager.h"
```

Include dependency graph for ASManager.cpp:

4.10 ASManager.cpp

Go to the documentation of this file.

```
00001 #include "ASManager.h"
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_scretch_size = 0;
00063     VkDeviceSize current_size = 0;
00064
00065     createAccelerationStructureInfosBLAS(device, build_as_structures[i],
00066                                          blas_input[i], current_scretch_size,
00067                                          current_size);
00068
00069     total_size_all_BLAS += current_size;
00070     max_scratch_size = std::max(max_scratch_size, current_scretch_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& 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_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_scretch_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_scretch_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 ito 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 }
```

4.11 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/renderer/↩ CommandBufferManager.cpp File Reference

`#include "CommandBufferManager.h"`

Include dependency graph for CommandBufferManager.cpp:

4.12 CommandBufferManager.cpp

Go to the documentation of this file.

```
00001 #include "CommandBufferManager.h"
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   // infromation 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() {}
```

4.13 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/renderer/PathTracing.cpp File Reference

```
#include "PathTracing.h"
#include <algorithm>
#include <array>
#include "File.h"
#include "ShaderHelper.h"
```

Include dependency graph for PathTracing.cpp:

4.14 PathTracing.cpp

Go to the documentation of this file.

```
00001 #include "PathTracing.h"
00002
00003 #include <algorithm>
00004 #include <array>
00005
00006 #include "File.h"
00007 #include "ShaderHelper.h"
00008
00009 // Good source:
00010 // https://github.com/nvpro-samples/vk_mini_path_tracer/blob/main/vk_mini_path_tracer/main.cpp
00011
00012 PathTracing::PathTracing() {}
00013
00014 void PathTracing::init(
00015     VulkanDevice* device,
00016     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00017   this->device = device;
00018
00019   VkPhysicalDeviceProperties physicalDeviceProps =
00020       device->getPhysicalDeviceProperties();
00021   timeStampPeriod = physicalDeviceProps.limits.timestampPeriod;
00022
00023   // save the limits for handling all special cases later on
00024   computeLimits.maxComputeWorkGroupCount[0] =
00025       physicalDeviceProps.limits.maxComputeWorkGroupCount[0];
00026   computeLimits.maxComputeWorkGroupCount[1] =
00027       physicalDeviceProps.limits.maxComputeWorkGroupCount[1];
00028   computeLimits.maxComputeWorkGroupCount[2] =
00029       physicalDeviceProps.limits.maxComputeWorkGroupCount[2];
00030
00031   computeLimits.maxComputeWorkGroupInvocations =
00032       physicalDeviceProps.limits.maxComputeWorkGroupInvocations;
00033
00034   computeLimits.maxComputeWorkGroupSize[0] =
00035       physicalDeviceProps.limits.maxComputeWorkGroupSize[0];
00036   computeLimits.maxComputeWorkGroupSize[1] =
```

```
00037       physicalDeviceProps.limits.maxComputeWorkGroupSize[1];
00038   computeLimits.maxComputeWorkGroupSize[2] =
00039       physicalDeviceProps.limits.maxComputeWorkGroupSize[2];
00040
00041   queryResults.resize(query_count);
00042   createQueryPool();
00043
00044   createPipeline(descriptorSetLayouts);
00045 }
00046
00047 void PathTracing::shaderHotReload(
00048     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
00049   vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
00050   createPipeline(descriptor_set_layouts);
00051 }
00052
00053 void PathTracing::recordCommands(
00054     VkCommandBuffer& commandBuffer, uint32_t image_index,
00055     VulkanImage& vulkanImage, VulkanSwapChain* vulkanSwapChain,
00056     const std::vector<VkDescriptorSet>& descriptorSets) {
00057   // we have reset the pool; hence start by 0
00058   uint32_t query = 0;
00059
00060   vkCmdResetQueryPool(commandBuffer, queryPool, 0, query_count);
00061
00062   vkCmdWriteTimestamp(
00063       commandBuffer,
00064       VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,
00065       query++);
00066
00067   QueueFamilyIndices indices = device->getQueueFamilies();
00068
00069   VkImageSubresourceRange subresourceRange{};
00070   subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
00071   subresourceRange.baseMipLevel = 0;
00072   subresourceRange.baseArrayLayer = 0;
00073   subresourceRange.levelCount = 1;
00074   subresourceRange.layerCount = 1;
00075
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
00093                        0, 0, nullptr, 0, nullptr, 1,
00094                        &presentToPathTracingImageBarrier);
00095
00096   VkExtent2D imageSize = vulkanSwapChain->getSwapChainExtent();
00097   push_constant.width = imageSize.width;
00098   push_constant.height = imageSize.height;
00099   push_constant.clearColor = {0.2f, 0.65f, 0.4f, 1.0f};
00100
00101   vkCmdPushConstants(commandBuffer, pipeline_layout,
00102                      VK_SHADER_STAGE_COMPUTE_BIT, 0,
00103                      sizeof(PushConstantPathTracing), &push_constant);
00104
00105   vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);
00106
00107   vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE,
00108                           pipeline_layout, 0,
00109                           static_cast<uint32_t>(descriptorSets.size()),
00110                           descriptorSets.data(), 0, 0);
00111
00112   uint32_t workGroupCountX =
00113       std::max((imageSize.width + specializationData.specWorkGroupSizeX - 1) /
00114                    specializationData.specWorkGroupSizeX,
00115                1U);
00116   uint32_t workGroupCountY =
00117       std::max((imageSize.height + specializationData.specWorkGroupSizeY - 1) /
00118                    specializationData.specWorkGroupSizeY,
00119                1U);
00120   uint32_t workGroupCountZ = 1;
00121
00122   vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY,
00123                 workGroupCountZ);
```

```
00124
00125   VkImageMemoryBarrier pathTracingToPresentImageBarrier{};
00126   pathTracingToPresentImageBarrier.sType =
00127       VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
00128   pathTracingToPresentImageBarrier.pNext = nullptr;
00129   pathTracingToPresentImageBarrier.srcQueueFamilyIndex = indices.compute_family;
00130   pathTracingToPresentImageBarrier.dstQueueFamilyIndex =
00131       indices.graphics_family;
00132   pathTracingToPresentImageBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
00133   pathTracingToPresentImageBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
00134   pathTracingToPresentImageBarrier.oldLayout = VK_IMAGE_LAYOUT_GENERAL;
00135   pathTracingToPresentImageBarrier.newLayout = VK_IMAGE_LAYOUT_GENERAL;
00136   pathTracingToPresentImageBarrier.image = vulkanImage.getImage();
00137   pathTracingToPresentImageBarrier.subresourceRange = subresourceRange;
00138
00139   vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
00140                        VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, 0, 0, nullptr, 0,
00141                        nullptr, 1, &pathTracingToPresentImageBarrier);
00142
00143   vkCmdWriteTimestamp(
00144       commandBuffer,
00145       VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,
00146       query++);
00147   VkResult result = vkGetQueryPoolResults(
00148       device->getLogicalDevice(), queryPool, 0, query_count,
00149       queryResults.size() * sizeof(uint64_t), queryResults.data(),
00150       static_cast<VkDeviceSize>(sizeof(uint64_t)), VK_QUERY_RESULT_64_BIT);
00151
00152   if (result != VK_NOT_READY) {
00153     pathTracingTiming = (static_cast<float>(queryResults[1] - queryResults[0]) *
00154                          timeStampPeriod) /
00155                         1000000.f;
00156   }
00157 }
00158
00159 void PathTracing::cleanUp() {
00160   vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
00161   vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00162
00163   vkDestroyQueryPool(device->getLogicalDevice(), queryPool, nullptr);
00164 }
00165
00166 PathTracing::~PathTracing() {}
00167
00168 void PathTracing::createQueryPool() {
00169   VkQueryPoolCreateInfo queryPoolInfo = {};
00170   queryPoolInfo.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
00171   // This query pool will store pipeline statistics
00172   queryPoolInfo.queryType = VK_QUERY_TYPE_TIMESTAMP;
00173   // Pipeline counters to be returned for this pool
00174   queryPoolInfo.pipelineStatistics =
00175       VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT;
00176   queryPoolInfo.queryCount = query_count;
00177   ASSERT_VULKAN(vkCreateQueryPool(device->getLogicalDevice(), &queryPoolInfo,
00178                                   NULL, &queryPool),
00179                 "Failed to create query pool!");
00180 }
00181
00182 void PathTracing::createPipeline(
00183     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00184   VkPushConstantRange push_constant_range{};
00185   push_constant_range.stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
00186   push_constant_range.offset = 0;
00187   push_constant_range.size = sizeof(PushConstantPathTracing);
00188
00189   VkPipelineLayoutCreateInfo compute_pipeline_layout_create_info{};
00190   compute_pipeline_layout_create_info.sType =
00191       VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00192   compute_pipeline_layout_create_info.setLayoutCount =
00193       static_cast<uint32_t>(descriptorSetLayouts.size());
00194   compute_pipeline_layout_create_info.pushConstantRangeCount = 1;
00195   compute_pipeline_layout_create_info.pPushConstantRanges =
00196       &push_constant_range;
00197   compute_pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00198
00199   ASSERT_VULKAN(vkCreatePipelineLayout(device->getLogicalDevice(),
00200                                        &compute_pipeline_layout_create_info,
00201                                        nullptr, &pipeline_layout),
00202                 "Failed to create compute path tracing pipeline layout!");
00203
00204   // create pipeline
00205   std::stringstream pathTracing_shader_dir;
00206   pathTracing_shader_dir « CMAKELISTS_DIR;
00207   pathTracing_shader_dir « "/Resources/Shader/path_tracing/";
00208
00209   std::string pathTracing_shader = "path_tracing.comp";
00210
```

```
00211   ShaderHelper shaderHelper;
00212   File pathTracingShaderFile(shaderHelper.getShaderSpvDir(
00213       pathTracing_shader_dir.str(), pathTracing_shader));
00214   std::vector<char> pathTracingShadercode =
00215       pathTracingShaderFile.readCharSequence();
00216
00217   shaderHelper.compileShader(pathTracing_shader_dir.str(), pathTracing_shader);
00218
00219   // build shader modules to link to graphics pipeline
00220   VkShaderModule pathTracingModule =
00221       shaderHelper.createShaderModule(device, pathTracingShadercode);
00222
00223   // Specialization constant for workgroup size
00224   std::array<VkSpecializationMapEntry, 2> specEntries{};
00225
00226   specEntries[0].constantID = 0;
00227   specEntries[0].size = sizeof(specializationData.specWorkGroupSizeX);
00228   specEntries[0].offset = 0;
00229
00230   specEntries[1].constantID = 1;
00231   specEntries[1].size = sizeof(specializationData.specWorkGroupSizeY);
00232   specEntries[1].offset = offsetof(SpecializationData, specWorkGroupSizeY);
00233
00234   // specEntries[2].constantID = 2;
00235   // specEntries[2].size = sizeof(specializationData.specWorkGroupSizeZ);
00236   // specEntries[2].offset = offsetof(SpecializationData, specWorkGroupSizeZ);
00237
00238   VkSpecializationInfo specInfo{};
00239   specInfo.dataSize = sizeof(specializationData);
00240   specInfo.mapEntryCount = static_cast<uint32_t>(specEntries.size());
00241   specInfo.pMapEntries = specEntries.data();
00242   specInfo.pData = &specializationData;
00243
00244   VkPipelineShaderStageCreateInfo compute_shader_integrate_create_info{};
00245   compute_shader_integrate_create_info.sType =
00246       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00247   compute_shader_integrate_create_info.stage = VK_SHADER_STAGE_COMPUTE_BIT;
00248   compute_shader_integrate_create_info.module = pathTracingModule;
00249   compute_shader_integrate_create_info.pSpecializationInfo = &specInfo;
00250   compute_shader_integrate_create_info.pName = "main";
00251
00252   // -- COMPUTE PIPELINE CREATION --
00253   VkComputePipelineCreateInfo compute_pipeline_create_info{};
00254   compute_pipeline_create_info.sType =
00255       VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
00256   compute_pipeline_create_info.stage = compute_shader_integrate_create_info;
00257   compute_pipeline_create_info.layout = pipeline_layout;
00258   compute_pipeline_create_info.flags = 0;
00259   // create compute pipeline
00260   ASSERT_VULKAN(vkCreateComputePipelines(
00261                     device->getLogicalDevice(), VK_NULL_HANDLE, 1,
00262                     &compute_pipeline_create_info, nullptr, &pipeline),
00263                 "Failed to create a compute pipeline!");
00264
00265   // Destroy shader modules, no longer needed after pipeline created
00266   vkDestroyShaderModule(device->getLogicalDevice(), pathTracingModule, nullptr);
00267 }
```

4.15 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/renderer/PostStage.cpp File Reference

```
#include "PostStage.h"
#include <array>
#include <vector>
#include "File.h"
#include "FormatHelper.h"
#include "GUI.h"
#include "ShaderHelper.h"
#include "Vertex.h"
```

Include dependency graph for PostStage.cpp:

4.16 PostStage.cpp

Go to the documentation of this file.

```
00001 #include "PostStage.h"
00002
00003 #include <array>
00004 #include <vector>
00005
00006 #include "File.h"
00007 #include "FormatHelper.h"
00008 #include "GUI.h"
00009 #include "ShaderHelper.h"
00010 #include "Vertex.h"
00011
00012 PostStage::PostStage() {}
00013
00014 void PostStage::init(
00015     VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
00016     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00017   this->device = device;
00018   this->vulkanSwapChain = vulkanSwapChain;
00019
00020   createOffscreenTextureSampler();
00021
00022   createPushConstantRange();
00023   createDepthbufferImage();
00024   createRenderpass();
00025   createGraphicsPipeline(descriptorSetLayouts);
00026   createFramebuffer();
00027 }
00028
00029 void PostStage::shaderHotReload(
00030     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
00031   vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00032   createGraphicsPipeline(descriptor_set_layouts);
00033 }
00034
00035 void PostStage::recordCommands(
00036     VkCommandBuffer& commandBuffer, uint32_t image_index,
00037     const std::vector<VkDescriptorSet>& descriptorSets) {
00038   // information about how to begin a render pass (only needed for graphical
00039   // applications)
00040   VkRenderPassBeginInfo render_pass_begin_info{};
00041   render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
00042   render_pass_begin_info.renderPass = render_pass;  // render pass to begin
00043   render_pass_begin_info.renderArea.offset = {
00044       0, 0};  // start point of render pass in pixels
00045   const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00046   render_pass_begin_info.renderArea.extent =
00047       swap_chain_extent;  // size of region to run render pass on (starting at
00048                           // offset)
00049
00050   // make sure the order you put the values into the array matches with the
00051   // attchment order you have defined previous
00052   std::array<VkClearValue, 2> clear_values = {};
00053   clear_values[0].color = {0.2f, 0.65f, 0.4f, 1.0f};
00054   clear_values[1].depthStencil = {1.0f, 0};
00055
00056   render_pass_begin_info.pClearValues = clear_values.data();
00057   render_pass_begin_info.clearValueCount =
00058       static_cast<uint32_t>(clear_values.size());
00059
00060   // used framebuffer depends on the swap chain and therefore is changing for
00061   // each command buffer
00062   render_pass_begin_info.framebuffer = framebuffers[image_index];
00063
00064   // begin render pass
00065   vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
00066                        VK_SUBPASS_CONTENTS_INLINE);
00067   auto aspectRatio = static_cast<float>(swap_chain_extent.width) /
00068                      static_cast<float>(swap_chain_extent.height);
00069   PushConstantPost pc_post{};
00070   pc_post.aspect_ratio = aspectRatio;
00071   vkCmdPushConstants(commandBuffer, pipeline_layout,
00072                      VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT,
00073                      0, sizeof(PushConstantPost), &pc_post);
00074   vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00075                     graphics_pipeline);
00076   vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00077                           pipeline_layout, 0,
00078                           static_cast<uint32_t>(descriptorSets.size()),
00079                           descriptorSets.data(), 0, nullptr);
00080   vkCmdDraw(commandBuffer, 3, 1, 0, 0);
00081
00082   // Rendering gui
00083   ImGui::Render();
00084   ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), commandBuffer);
00085
00086   // end render pass
00087   vkCmdEndRenderPass(commandBuffer);
```

```
00088 }
00089
00090 void PostStage::cleanUp() {
00091   depthBufferImage.cleanUp();
00092   for (auto framebuffer : framebuffers) {
00093     vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
00094   }
00095
00096   vkDestroySampler(device->getLogicalDevice(), offscreenTextureSampler,
00097                    nullptr);
00098
00099   vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
00100   vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00101   vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00102 }
00103
00104 PostStage::~PostStage() {}
00105
00106 void PostStage::createDepthbufferImage() {
00107   // get supported format for depth buffer
00108   depth_format = choose_supported_format(
00109       device->getPhysicalDevice(),
00110       {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00111        VK_FORMAT_D24_UNORM_S8_UINT},
00112       VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00113
00114   // create depth buffer image
00115   // MIP LEVELS: for depth texture we only want 1 level :)
00116   const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00117   depthBufferImage.createImage(device, swap_chain_extent.width,
00118                                swap_chain_extent.height, 1, depth_format,
00119                                VK_IMAGE_TILING_OPTIMAL,
00120                                VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
00121                                VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00122
00123   // depth buffer image view
00124   // MIP LEVELS: for depth texture we only want 1 level :)
00125   depthBufferImage.createImageView(device, depth_format,
00126                                    VK_IMAGE_ASPECT_DEPTH_BIT, 1);
00127 }
00128
00129 void PostStage::createOffscreenTextureSampler() {
00130   // sampler create info
00131   VkSamplerCreateInfo sampler_create_info{};
00132   sampler_create_info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
00133   sampler_create_info.magFilter = VK_FILTER_LINEAR;
00134   sampler_create_info.minFilter = VK_FILTER_LINEAR;
00135   sampler_create_info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00136   sampler_create_info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00137   sampler_create_info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
00138   sampler_create_info.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
00139   sampler_create_info.unnormalizedCoordinates = VK_FALSE;
00140   sampler_create_info.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
00141   sampler_create_info.mipLodBias = 0.0f;
00142   sampler_create_info.minLod = 0.0f;
00143   sampler_create_info.maxLod = 0.0f;
00144   sampler_create_info.anisotropyEnable = VK_TRUE;
00145   sampler_create_info.maxAnisotropy = 16;  // max anisotropy sample level
00146
00147   VkResult result =
00148       vkCreateSampler(device->getLogicalDevice(), &sampler_create_info, nullptr,
00149                       &offscreenTextureSampler);
00150   ASSERT_VULKAN(result, "Failed to create a texture sampler!")
00151 }
00152
00153 void PostStage::createPushConstantRange() {
00154   push_constant_range.stageFlags =
00155       VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
00156   push_constant_range.offset = 0;
00157   push_constant_range.size = sizeof(PushConstantPost);
00158 }
00159
00160 void PostStage::createRenderpass() {
00161   // Color attachment of render pass
00162   VkAttachmentDescription color_attachment{};
00163   const VkFormat& swap_chain_image_format =
00164       vulkanSwapChain->getSwapChainFormat();
00165   color_attachment.format =
00166       swap_chain_image_format;  // format to use for attachment
00167   color_attachment.samples =
00168       VK_SAMPLE_COUNT_1_BIT;  // number of samples to write for multisampling
00169   color_attachment.loadOp =
00170       VK_ATTACHMENT_LOAD_OP_CLEAR;  // describes what to do with attachment
00171                                     // before rendering
00172   color_attachment.storeOp =
00173       VK_ATTACHMENT_STORE_OP_STORE;  // describes what to do with attachment
00174                                      // after rendering
```

```
00175   color_attachment.stencilLoadOp =
00176       VK_ATTACHMENT_LOAD_OP_DONT_CARE;  // describes what to do with stencil
00177                                         // before rendering
00178   color_attachment.stencilStoreOp =
00179       VK_ATTACHMENT_STORE_OP_DONT_CARE;  // describes what to do with stencil
00180                                          // after rendering
00181
00182   // framebuffer data will be stored as an image, but images can be given
00183   // different layouts to give optimal use for certain operations
00184   color_attachment.initialLayout =
00185       VK_IMAGE_LAYOUT_UNDEFINED;  // image data layout before render pass starts
00186   color_attachment.finalLayout =
00187       VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;  // image data layout after render pass
00188                                         // (to change to)
00189
00190   // depth attachment of render pass
00191   VkAttachmentDescription depth_attachment{};
00192   depth_attachment.format = choose_supported_format(
00193       device->getPhysicalDevice(),
00194       {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00195        VK_FORMAT_D24_UNORM_S8_UINT},
00196       VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00197   depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
00198   depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
00199   depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00200   depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
00201   depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00202   depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
00203   depth_attachment.finalLayout =
00204       VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00205
00206   // attachment reference uses an attachment index that refers to index in the
00207   // attachment list passed to renderPassCreateInfo
00208   VkAttachmentReference color_attachment_reference{};
00209   color_attachment_reference.attachment = 0;
00210   color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
00211
00212   // attachment reference
00213   VkAttachmentReference depth_attachment_reference{};
00214   depth_attachment_reference.attachment = 1;
00215   depth_attachment_reference.layout =
00216       VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00217
00218   // information about a particular subpass the render pass is using
00219   VkSubpassDescription subpass{};
00220   subpass.pipelineBindPoint =
00221       VK_PIPELINE_BIND_POINT_GRAPHICS;  // pipeline type subpass is to be bound
00222                                         // to
00223   subpass.colorAttachmentCount = 1;
00224   subpass.pColorAttachments = &color_attachment_reference;
00225   subpass.pDepthStencilAttachment = &depth_attachment_reference;
00226
00227   // need to determine when layout transitions occur using subpass dependencies
00228   std::array<VkSubpassDependency, 1> subpass_dependencies;
00229
00230   // conversion from VK_IMAGE_LAYOUT_UNDEFINED to
00231   // VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
00232   subpass_dependencies[0].srcSubpass =
00233       VK_SUBPASS_EXTERNAL;  // subpass index (VK_SUBPASS_EXTERNAL = Special
00234                             // value meaning outside of renderpass)
00235   subpass_dependencies[0].srcStageMask =
00236       VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;  // pipeline stage
00237   subpass_dependencies[0].srcAccessMask =
00238       VK_ACCESS_MEMORY_READ_BIT;  // stage access mask (memory access)
00239   subpass_dependencies[0].dstSubpass = 0;
00240   subpass_dependencies[0].dstStageMask =
00241       VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00242   subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
00243                                           VK_ACCESS_COLOR_ATTACHMENT_READ_BIT;
00244   subpass_dependencies[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
00245
00246   std::array<VkAttachmentDescription, 2> render_pass_attachments = {
00247       color_attachment, depth_attachment};
00248
00249   // create info for render pass
00250   VkRenderPassCreateInfo render_pass_create_info{};
00251   render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
00252   render_pass_create_info.attachmentCount =
00253       static_cast<uint32_t>(render_pass_attachments.size());
00254   render_pass_create_info.pAttachments = render_pass_attachments.data();
00255   render_pass_create_info.subpassCount = 1;
00256   render_pass_create_info.pSubpasses = &subpass;
00257   render_pass_create_info.dependencyCount =
00258       static_cast<uint32_t>(subpass_dependencies.size());
00259   render_pass_create_info.pDependencies = subpass_dependencies.data();
00260
00261   VkResult result =
```

```
00262       vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
00263                          nullptr, &render_pass);
00264   ASSERT_VULKAN(result, "Failed to create render pass!")
00265 }
00266
00267 void PostStage::createGraphicsPipeline(
00268     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00269   std::stringstream post_shader_dir;
00270   post_shader_dir « CMAKELISTS_DIR;
00271   post_shader_dir « "/Resources/Shader/post/";
00272
00273   std::string post_vert_shader = "post.vert";
00274   std::string post_frag_shader = "post.frag";
00275
00276   ShaderHelper shaderHelper;
00277   File vertexShaderFile(
00278       shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_vert_shader));
00279   std::vector<char> vertex_shader_code = vertexShaderFile.readCharSequence();
00280   File fragmentShaderFile(
00281       shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_frag_shader));
00282   std::vector<char> fragment_shader_code =
00283       fragmentShaderFile.readCharSequence();
00284
00285   shaderHelper.compileShader(post_shader_dir.str(), post_vert_shader);
00286   shaderHelper.compileShader(post_shader_dir.str(), post_frag_shader);
00287
00288   // build shader modules to link to graphics pipeline
00289   VkShaderModule vertex_shader_module =
00290       shaderHelper.createShaderModule(device, vertex_shader_code);
00291   VkShaderModule fragment_shader_module =
00292       shaderHelper.createShaderModule(device, fragment_shader_code);
00293
00294   // shader stage creation information
00295   // vertex stage creation information
00296   VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
00297   vertex_shader_create_info.sType =
00298       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00299   vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
00300   vertex_shader_create_info.module = vertex_shader_module;
00301   vertex_shader_create_info.pName = "main";
00302
00303   // fragment stage creation information
00304   VkPipelineShaderStageCreateInfo fragment_shader_create_info{};
00305   fragment_shader_create_info.sType =
00306       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00307   fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
00308   fragment_shader_create_info.module = fragment_shader_module;
00309   fragment_shader_create_info.pName = "main";
00310
00311   std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
00312       vertex_shader_create_info, fragment_shader_create_info};
00313
00314   // how the data for a single vertex (including info such as position, color,
00315   // texture coords, normals, etc) is as a whole
00316   VkVertexInputBindingDescription binding_description{};
00317   binding_description.binding = 0;
00318   binding_description.stride = sizeof(Vertex);
00319   binding_description.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
00320
00321   std::array<VkVertexInputAttributeDescription, 4> attribute_describtions =
00322       vertex::getVertexInputAttributeDesc();
00323
00324   // CREATE PIPELINE
00325   // 1.) Vertex input
00326   VkPipelineVertexInputStateCreateInfo vertex_input_create_info{};
00327   vertex_input_create_info.sType =
00328       VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
00329   vertex_input_create_info.vertexBindingDescriptionCount = 0;
00330   vertex_input_create_info.pVertexBindingDescriptions = nullptr;
00331   vertex_input_create_info.vertexAttributeDescriptionCount = 0;
00332   vertex_input_create_info.pVertexAttributeDescriptions = nullptr;
00333
00334   // input assembly
00335   VkPipelineInputAssemblyStateCreateInfo input_assembly{};
00336   input_assembly.sType =
00337       VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
00338   input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
00339   input_assembly.primitiveRestartEnable = VK_FALSE;
00340
00341   // viewport & scissor
00342   // create a viewport info struct
00343   VkViewport viewport{};
00344   viewport.x = 0.0f;
00345   viewport.y = 0.0f;
00346   const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00347   viewport.width = (float)swap_chain_extent.width;
00348   viewport.height = (float)swap_chain_extent.height;
```

```
00349   viewport.minDepth = 0.0f;
00350   viewport.maxDepth = 1.0f;
00351
00352   // create a scissor info struct
00353   VkRect2D scissor{};
00354   scissor.offset = {0, 0};
00355   scissor.extent = swap_chain_extent;
00356
00357   VkPipelineViewportStateCreateInfo viewport_state_create_info{};
00358   viewport_state_create_info.sType =
00359       VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
00360   viewport_state_create_info.viewportCount = 1;
00361   viewport_state_create_info.pViewports = &viewport;
00362   viewport_state_create_info.scissorCount = 1;
00363   viewport_state_create_info.pScissors = &scissor;
00364
00365   // RASTERIZER
00366   VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
00367   rasterizer_create_info.sType =
00368       VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
00369   rasterizer_create_info.depthClampEnable = VK_FALSE;
00370   rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
00371   rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
00372   rasterizer_create_info.lineWidth = 1.0f;
00373   rasterizer_create_info.cullMode = VK_CULL_MODE_NONE;
00374   // winding to determine which side is front; y-coordinate is inverted in
00375   // comparison to OpenGL
00376   rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
00377   rasterizer_create_info.depthBiasClamp = VK_FALSE;
00378
00379   // -- MULTISAMPLING --
00380   VkPipelineMultisampleStateCreateInfo multisample_create_info{};
00381   multisample_create_info.sType =
00382       VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
00383   multisample_create_info.sampleShadingEnable = VK_FALSE;
00384   multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
00385
00386   // -- BLENDING --
00387   // blend attachment state
00388   VkPipelineColorBlendAttachmentState color_state{};
00389   color_state.colorWriteMask =
00390       VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
00391       VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
00392
00393   color_state.blendEnable = VK_TRUE;
00394   // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
00395   // (dstColorBlendFactor * old_color)
00396   color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
00397   color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
00398   color_state.colorBlendOp = VK_BLEND_OP_ADD;
00399   color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
00400   color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
00401   color_state.alphaBlendOp = VK_BLEND_OP_ADD;
00402
00403   VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
00404   color_blending_create_info.sType =
00405       VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
00406   color_blending_create_info.logicOpEnable =
00407       VK_FALSE;  // alternative to calculations is to use logical operations
00408   color_blending_create_info.logicOp = VK_LOGIC_OP_CLEAR;
00409   color_blending_create_info.attachmentCount = 1;
00410   color_blending_create_info.pAttachments = &color_state;
00411   for (int i = 0; i < 4; i++) {
00412     color_blending_create_info.blendConstants[0] = 0.f;
00413   }
00414   // -- PIPELINE LAYOUT --
00415   VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00416   pipeline_layout_create_info.sType =
00417       VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00418   pipeline_layout_create_info.setLayoutCount =
00419       static_cast<uint32_t>(descriptorSetLayouts.size());
00420   pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00421   pipeline_layout_create_info.pushConstantRangeCount = 1;
00422   pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;
00423
00424   // create pipeline layout
00425   VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00426                                            &pipeline_layout_create_info,
00427                                            nullptr, &pipeline_layout);
00428   ASSERT_VULKAN(result, "Failed to create pipeline layout!")
00429
00430   // -- DEPTH STENCIL TESTING --
00431   VkPipelineDepthStencilStateCreateInfo depth_stencil_create_info{};
00432   depth_stencil_create_info.sType =
00433       VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
00434   depth_stencil_create_info.depthTestEnable = VK_TRUE;
00435   depth_stencil_create_info.depthWriteEnable = VK_TRUE;
```

```
00436   depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
00437   depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
00438   depth_stencil_create_info.stencilTestEnable = VK_FALSE;
00439
00440   // -- GRAPHICS PIPELINE CREATION --
00441   VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
00442   graphics_pipeline_create_info.sType =
00443       VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
00444   graphics_pipeline_create_info.stageCount =
00445       static_cast<uint32_t>(shader_stages.size());
00446   graphics_pipeline_create_info.pStages = shader_stages.data();
00447   graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
00448   graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
00449   graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
00450   graphics_pipeline_create_info.pDynamicState = nullptr;
00451   graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
00452   graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
00453   graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
00454   graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
00455   graphics_pipeline_create_info.layout = pipeline_layout;
00456   graphics_pipeline_create_info.renderPass = render_pass;
00457   graphics_pipeline_create_info.subpass = 0;
00458
00459   // pipeline derivatives : can create multiple pipelines that derive from one
00460   // another for optimization
00461   graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
00462   graphics_pipeline_create_info.basePipelineIndex = -1;
00463
00464   // create graphics pipeline
00465   result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
00466                                      1, &graphics_pipeline_create_info, nullptr,
00467                                      &graphics_pipeline);
00468   ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")
00469
00470   // Destroy shader modules, no longer needed after pipeline created
00471   vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
00472                         nullptr);
00473   vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,
00474                         nullptr);
00475 }
00476
00477 void PostStage::createFramebuffer() {
00478   // resize framebuffer size to equal swap chain image count
00479   framebuffers.resize(vulkanSwapChain->getNumberSwapChainImages());
00480
00481   for (size_t i = 0; i < vulkanSwapChain->getNumberSwapChainImages(); i++) {
00482     Texture& swap_chain_image = vulkanSwapChain->getSwapChainImage(i);
00483
00484     std::array<VkImageView, 2> attachments = {swap_chain_image.getImageView(),
00485                                               depthBufferImage.getImageView()};
00486
00487     VkFramebufferCreateInfo frame_buffer_create_info{};
00488     frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
00489     frame_buffer_create_info.renderPass =
00490         render_pass;  // render pass layout the framebuffer will be used with
00491     frame_buffer_create_info.attachmentCount =
00492         static_cast<uint32_t>(attachments.size());
00493     frame_buffer_create_info.pAttachments =
00494         attachments.data();  // list of attachments (1:1 with render pass)
00495     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00496     frame_buffer_create_info.width =
00497         swap_chain_extent.width;  // framebuffer width
00498     frame_buffer_create_info.height =
00499         swap_chain_extent.height;         // framebuffer height
00500     frame_buffer_create_info.layers = 1;  // framebuffer layer
00501
00502     VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
00503                                           &frame_buffer_create_info, nullptr,
00504                                           &framebuffers[i]);
00505     ASSERT_VULKAN(result, "Failed to create framebuffer!")
00506   }
00507 }
```

4.17 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/renderer/↩ Rasterizer.cpp File Reference

```
#include "Rasterizer.h"
#include <array>
#include <vector>
```

```
#include "File.h"
#include "FormatHelper.h"
#include "ShaderHelper.h"
#include "Vertex.h"
```

Include dependency graph for Rasterizer.cpp:

4.18 Rasterizer.cpp

Go to the documentation of this file.

```
00001 #include "Rasterizer.h"
00002
00003 #include <array>
00004 #include <vector>
00005
00006 #include "File.h"
00007 #include "FormatHelper.h"
00008 #include "ShaderHelper.h"
00009 #include "Vertex.h"
00010
00011 Rasterizer::Rasterizer() {}
00012
00013 void Rasterizer::init(
00014     VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
00015     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts,
00016     VkCommandPool& commandPool) {
00017   this->device = device;
00018   this->vulkanSwapChain = vulkanSwapChain;
00019
00020   createTextures(commandPool);
00021   createRenderPass();
00022   createPushConstantRange();
00023   createGraphicsPipeline(descriptorSetLayouts);
00024   createFramebuffer();
00025 }
00026
00027 void Rasterizer::shaderHotReload(
00028     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
00029   vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00030   createGraphicsPipeline(descriptor_set_layouts);
00031 }
00032
00033 Texture& Rasterizer::getOffscreenTexture(uint32_t index) {
00034   return offscreenTextures[index];
00035 }
00036
00037 void Rasterizer::setPushConstant(PushConstantRasterizer pushConstant) {
00038   this->pushConstant = pushConstant;
00039 }
00040
00041 void Rasterizer::recordCommands(
00042     VkCommandBuffer& commandBuffer, uint32_t image_index, Scene* scene,
00043     const std::vector<VkDescriptorSet>& descriptorSets) {
00044   // information about how to begin a render pass (only needed for graphical
00045   // applications)
00046   VkRenderPassBeginInfo render_pass_begin_info{};
00047   render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
00048   render_pass_begin_info.renderPass = render_pass;
00049   render_pass_begin_info.renderArea.offset = {0, 0};
00050   const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00051   render_pass_begin_info.renderArea.extent = swap_chain_extent;
00052
00053   // make sure the order you put the values into the array matches with the
00054   // attchment 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   render_pass_begin_info.framebuffer = framebuffer[image_index];
00063
00064   // begin render pass
00065   vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
00066                        VK_SUBPASS_CONTENTS_INLINE);
00067
00068   // bind pipeline to be used in render pass
00069   vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00070                     graphics_pipeline);
```

```
00071
00072   for (uint32_t m = 0; m < static_cast<uint32_t>(scene->getModelCount()); m++) {
00073     // for GCC doen't allow references on rvalues go like that ...
00074     pushConstant.model = scene->getModelMatrix(0);
00075     // just "Push" constants to given shader stage directly (no buffer)
00076     vkCmdPushConstants(
00077         commandBuffer, pipeline_layout,
00078         VK_SHADER_STAGE_VERTEX_BIT,      // stage to push constants to
00079         0,                               // offset to push constants to update
00080         sizeof(PushConstantRasterizer),  // size of data being pushed
00081         &pushConstant);  // using model of current mesh (can be array)
00082
00083     for (unsigned int k = 0; k < scene->getMeshCount(m); k++) {
00084       // list of vertex buffers we want to draw
00085       VkBuffer vertex_buffers[] = {
00086           scene->getVertexBuffer(m, k)};  // buffers to bind
00087       VkDeviceSize offsets[] = {0};
00088       vkCmdBindVertexBuffers(
00089           commandBuffer, 0, 1, vertex_buffers,
00090           offsets);  // command to bind vertex buffer before drawing with them
00091
00092       // bind mesh index buffer with 0 offset and using the uint32 type
00093       vkCmdBindIndexBuffer(commandBuffer, scene->getIndexBuffer(m, k), 0,
00094                            VK_INDEX_TYPE_UINT32);
00095
00096       // bind descriptor sets
00097       vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
00098                               pipeline_layout, 0,
00099                               static_cast<uint32_t>(descriptorSets.size()),
00100                               descriptorSets.data(), 0, nullptr);
00101
00102       // execute pipeline
00103       vkCmdDrawIndexed(commandBuffer,
00104                        static_cast<uint32_t>(scene->getIndexCount(m, k)), 1, 0,
00105                        0, 0);
00106     }
00107   }
00108
00109   // end render pass
00110   vkCmdEndRenderPass(commandBuffer);
00111 }
00112
00113 void Rasterizer::cleanUp() {
00114   for (auto framebuffer : framebuffer) {
00115     vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
00116   }
00117
00118   for (Texture texture : offscreenTextures) {
00119     texture.cleanUp();
00120   }
00121
00122   depthBufferImage.cleanUp();
00123
00124   vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
00125   vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00126   vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
00127 }
00128
00129 Rasterizer::~Rasterizer() {}
00130
00131 void Rasterizer::createRenderPass() {
00132   // Color attachment of render pass
00133   VkAttachmentDescription color_attachment{};
00134   const VkFormat& swap_chain_image_format =
00135       vulkanSwapChain->getSwapChainFormat();
00136   color_attachment.format =
00137       swap_chain_image_format;  // format to use for attachment
00138   color_attachment.samples =
00139       VK_SAMPLE_COUNT_1_BIT;  // number of samples to write for multisampling
00140   color_attachment.loadOp =
00141       VK_ATTACHMENT_LOAD_OP_CLEAR;  // describes what to do with attachment
00142                                     // before rendering
00143   color_attachment.storeOp =
00144       VK_ATTACHMENT_STORE_OP_STORE;  // describes what to do with attachment
00145                                      // after rendering
00146   color_attachment.stencilLoadOp =
00147       VK_ATTACHMENT_LOAD_OP_DONT_CARE;  // describes what to do with stencil
00148                                         // before rendering
00149   color_attachment.stencilStoreOp =
00150       VK_ATTACHMENT_STORE_OP_DONT_CARE;  // describes what to do with stencil
00151                                          // after rendering
00152
00153   // framebuffer data will be stored as an image, but images can be given
00154   // different layouts to give optimal use for certain operations
00155   color_attachment.initialLayout =
00156       VK_IMAGE_LAYOUT_GENERAL;  // image data layout before render pass starts
00157   color_attachment.finalLayout =
```

```
00158       VK_IMAGE_LAYOUT_GENERAL;  // image data layout after render pass (to
00159                                 // change to)
00160
00161   // depth attachment of render pass
00162   VkAttachmentDescription depth_attachment{};
00163   depth_attachment.format = choose_supported_format(
00164       device->getPhysicalDevice(),
00165       {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00166        VK_FORMAT_D24_UNORM_S8_UINT},
00167       VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00168
00169   depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
00170   depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
00171   depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00172   depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
00173   depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
00174   depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
00175   depth_attachment.finalLayout =
00176       VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00177
00178   // attachment reference uses an attachment index that refers to index in the
00179   // attachment list passed to renderPassCreateInfo
00180   VkAttachmentReference color_attachment_reference{};
00181   color_attachment_reference.attachment = 0;
00182   color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
00183
00184   // attachment reference
00185   VkAttachmentReference depth_attachment_reference{};
00186   depth_attachment_reference.attachment = 1;
00187   depth_attachment_reference.layout =
00188       VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
00189
00190   // information about a particular subpass the render pass is using
00191   VkSubpassDescription subpass{};
00192   subpass.pipelineBindPoint =
00193       VK_PIPELINE_BIND_POINT_GRAPHICS;  // pipeline type subpass is to be bound
00194                                         // to
00195   subpass.colorAttachmentCount = 1;
00196   subpass.pColorAttachments = &color_attachment_reference;
00197   subpass.pDepthStencilAttachment = &depth_attachment_reference;
00198
00199   // need to determine when layout transitions occur using subpass dependencies
00200   std::array<VkSubpassDependency, 1> subpass_dependencies;
00201
00202   // conversion from VK_IMAGE_LAYOUT_UNDEFINED to
00203   // VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
00204   subpass_dependencies[0].srcSubpass =
00205       VK_SUBPASS_EXTERNAL;  // subpass index (VK_SUBPASS_EXTERNAL = Special
00206                             // value meaning outside of renderpass)
00207   subpass_dependencies[0].srcStageMask =
00208       VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;  // pipeline stage
00209   subpass_dependencies[0].srcAccessMask =
00210       0;  // stage access mask (memory access)
00211
00212   // but must happen before ...
00213   subpass_dependencies[0].dstSubpass = 0;
00214   subpass_dependencies[0].dstStageMask =
00215       VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
00216   subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
00217   subpass_dependencies[0].dependencyFlags = 0;  // VK_DEPENDENCY_BY_REGION_BIT;
00218
00219   std::array<VkAttachmentDescription, 2> render_pass_attachments = {
00220       color_attachment, depth_attachment};
00221
00222   // create info for render pass
00223   VkRenderPassCreateInfo render_pass_create_info{};
00224   render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
00225   render_pass_create_info.attachmentCount =
00226       static_cast<uint32_t>(render_pass_attachments.size());
00227   render_pass_create_info.pAttachments = render_pass_attachments.data();
00228   render_pass_create_info.subpassCount = 1;
00229   render_pass_create_info.pSubpasses = &subpass;
00230   render_pass_create_info.dependencyCount =
00231       static_cast<uint32_t>(subpass_dependencies.size());
00232   render_pass_create_info.pDependencies = subpass_dependencies.data();
00233
00234   VkResult result =
00235       vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
00236                          nullptr, &render_pass);
00237   ASSERT_VULKAN(result, "Failed to create render pass!")
00238 }
00239
00240 void Rasterizer::createFramebuffer() {
00241   framebuffer.resize(vulkanSwapChain->getNumberSwapChainImages());
00242
00243   for (size_t i = 0; i < framebuffer.size(); i++) {
00244     std::array<VkImageView, 2> attachments = {
```

```
00245         offscreenTextures[i].getImageView(), depthBufferImage.getImageView()};
00246
00247     VkFramebufferCreateInfo frame_buffer_create_info{};
00248     frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
00249     frame_buffer_create_info.renderPass = render_pass;
00250     frame_buffer_create_info.attachmentCount =
00251         static_cast<uint32_t>(attachments.size());
00252     frame_buffer_create_info.pAttachments = attachments.data();
00253     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00254     frame_buffer_create_info.width = swap_chain_extent.width;
00255     frame_buffer_create_info.height = swap_chain_extent.height;
00256     frame_buffer_create_info.layers = 1;
00257
00258     VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
00259                                           &frame_buffer_create_info, nullptr,
00260                                           &framebuffer[i]);
00261     ASSERT_VULKAN(result, "Failed to create framebuffer!");
00262   }
00263 }
00264
00265 void Rasterizer::createPushConstantRange() {
00266   // define push constant values (no 'create' needed)
00267   push_constant_range.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
00268   push_constant_range.offset = 0;
00269   push_constant_range.size = sizeof(PushConstantRasterizer);
00270 }
00271
00272 void Rasterizer::createTextures(VkCommandPool& commandPool) {
00273   offscreenTextures.resize(vulkanSwapChain->getNumberSwapChainImages());
00274
00275   VkCommandBuffer cmdBuffer = commandBufferManager.beginCommandBuffer(
00276       device->getLogicalDevice(), commandPool);
00277
00278   for (uint32_t index = 0;
00279        index <
00280        static_cast<uint32_t>(vulkanSwapChain->getNumberSwapChainImages());
00281        index++) {
00282     Texture texture{};
00283     const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00284     const VkFormat& swap_chain_image_format =
00285         vulkanSwapChain->getSwapChainFormat();
00286
00287     texture.createImage(
00288         device, swap_chain_extent.width, swap_chain_extent.height, 1,
00289         swap_chain_image_format, VK_IMAGE_TILING_OPTIMAL,
00290         VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT |
00291             VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT,
00292         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00293
00294     texture.createImageView(device, swap_chain_image_format,
00295                             VK_IMAGE_ASPECT_COLOR_BIT, 1);
00296
00297     // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
00298     VulkanImage& image = texture.getVulkanImage();
00299     image.transitionImageLayout(cmdBuffer, VK_IMAGE_LAYOUT_UNDEFINED,
00300                                 VK_IMAGE_LAYOUT_GENERAL, 1,
00301                                 VK_IMAGE_ASPECT_COLOR_BIT);
00302
00303     offscreenTextures[index] = texture;
00304   }
00305
00306   VkFormat depth_format = choose_supported_format(
00307       device->getPhysicalDevice(),
00308       {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
00309        VK_FORMAT_D24_UNORM_S8_UINT},
00310       VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
00311
00312   // create depth buffer image
00313   // MIP LEVELS: for depth texture we only want 1 level :)
00314   const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00315   depthBufferImage.createImage(device, swap_chain_extent.width,
00316                                swap_chain_extent.height, 1, depth_format,
00317                                VK_IMAGE_TILING_OPTIMAL,
00318                                VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
00319                                VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
00320
00321   // depth buffer image view
00322   // MIP LEVELS: for depth texture we only want 1 level :)
00323   depthBufferImage.createImageView(
00324       device, depth_format,
00325       VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);
00326
00327   // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
00328   VulkanImage& vulkanImage = depthBufferImage.getVulkanImage();
00329   vulkanImage.transitionImageLayout(
00330       device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
00331       VK_IMAGE_LAYOUT_UNDEFINED,
```

```
00332       VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
00333       VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);
00334
00335   commandBufferManager.endAndSubmitCommandBuffer(
00336       device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
00337       cmdBuffer);
00338 }
00339
00340 void Rasterizer::createGraphicsPipeline(
00341     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00342   std::stringstream rasterizer_shader_dir;
00343   rasterizer_shader_dir « CMAKELISTS_DIR;
00344   rasterizer_shader_dir « "/Resources/Shader/rasterizer/";
00345
00346   ShaderHelper shaderHelper;
00347   shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.vert");
00348   shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.frag");
00349
00350   File vertexFile(
00351       shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.vert"));
00352   File fragmentFile(
00353       shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.frag"));
00354   std::vector<char> vertex_shader_code = vertexFile.readCharSequence();
00355   std::vector<char> fragment_shader_code = fragmentFile.readCharSequence();
00356
00357   // build shader modules to link to graphics pipeline
00358   VkShaderModule vertex_shader_module =
00359       shaderHelper.createShaderModule(device, vertex_shader_code);
00360   VkShaderModule fragment_shader_module =
00361       shaderHelper.createShaderModule(device, fragment_shader_code);
00362
00363   // shader stage creation information
00364   // vertex stage creation information
00365   VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
00366   vertex_shader_create_info.sType =
00367       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00368   vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
00369   vertex_shader_create_info.module = vertex_shader_module;
00370   vertex_shader_create_info.pName = "main";
00371
00372   // fragment stage creation information
00373   VkPipelineShaderStageCreateInfo fragment_shader_create_info{};
00374   fragment_shader_create_info.sType =
00375       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00376   fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
00377   fragment_shader_create_info.module = fragment_shader_module;
00378   fragment_shader_create_info.pName = "main";
00379
00380   std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
00381       vertex_shader_create_info, fragment_shader_create_info};
00382
00383   // how the data for a single vertex (including info such as position, color,
00384   // texture coords, normals, etc) is as a whole
00385   VkVertexInputBindingDescription binding_description{};
00386   binding_description.binding = 0;
00387   binding_description.stride = sizeof(Vertex);
00388   binding_description.inputRate =
00389       VK_VERTEX_INPUT_RATE_VERTEX;  // how to move between data after each
00390                                     // vertex.
00391
00392   // how the data for an attribute is defined within a vertex
00393   std::array<VkVertexInputAttributeDescription, 4> attribute_describtions =
00394       vertex::getVertexInputAttributeDesc();
00395
00396   // CREATE PIPELINE
00397   // 1.) Vertex input
00398   VkPipelineVertexInputStateCreateInfo vertex_input_create_info{};
00399   vertex_input_create_info.sType =
00400       VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
00401   vertex_input_create_info.vertexBindingDescriptionCount = 1;
00402   vertex_input_create_info.pVertexBindingDescriptions = &binding_description;
00403   vertex_input_create_info.vertexAttributeDescriptionCount =
00404       static_cast<uint32_t>(attribute_describtions.size());
00405   vertex_input_create_info.pVertexAttributeDescriptions =
00406       attribute_describtions.data();
00407
00408   // input assembly
00409   VkPipelineInputAssemblyStateCreateInfo input_assembly{};
00410   input_assembly.sType =
00411       VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
00412   input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
00413   input_assembly.primitiveRestartEnable = VK_FALSE;
00414
00415   // viewport & scissor
00416   // create a viewport info struct
00417   VkViewport viewport{};
00418   viewport.x = 0.0f;
```

```
00419   viewport.y = 0.0f;
00420   const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00421   viewport.width = (float)swap_chain_extent.width;
00422   viewport.height = (float)swap_chain_extent.height;
00423   viewport.minDepth = 0.0f;
00424   viewport.maxDepth = 1.0f;
00425
00426   // create a scissor info struct
00427   VkRect2D scissor{};
00428   scissor.offset = {0, 0};
00429   scissor.extent = swap_chain_extent;
00430
00431   VkPipelineViewportStateCreateInfo viewport_state_create_info{};
00432   viewport_state_create_info.sType =
00433       VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
00434   viewport_state_create_info.viewportCount = 1;
00435   viewport_state_create_info.pViewports = &viewport;
00436   viewport_state_create_info.scissorCount = 1;
00437   viewport_state_create_info.pScissors = &scissor;
00438
00439   // RASTERIZER
00440   VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
00441   rasterizer_create_info.sType =
00442       VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
00443   rasterizer_create_info.depthClampEnable = VK_FALSE;
00444   rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
00445   rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
00446   rasterizer_create_info.lineWidth = 1.0f;
00447   rasterizer_create_info.cullMode = VK_CULL_MODE_BACK_BIT;  //
00448   // winding to determine which side is front; y-coordinate is inverted in
00449   // comparison to OpenGL
00450   rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
00451   rasterizer_create_info.depthBiasClamp = VK_FALSE;
00452
00453   // -- MULTISAMPLING --
00454   VkPipelineMultisampleStateCreateInfo multisample_create_info{};
00455   multisample_create_info.sType =
00456       VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
00457   multisample_create_info.sampleShadingEnable = VK_FALSE;
00458   multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
00459
00460   // -- BLENDING --
00461   // blend attachment state
00462   VkPipelineColorBlendAttachmentState color_state{};
00463   color_state.colorWriteMask =
00464       VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
00465       VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
00466
00467   color_state.blendEnable = VK_TRUE;
00468   // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
00469   // (dstColorBlendFactor * old_color)
00470   color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
00471   color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
00472   color_state.colorBlendOp = VK_BLEND_OP_ADD;
00473   color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
00474   color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
00475   color_state.alphaBlendOp = VK_BLEND_OP_ADD;
00476
00477   VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
00478   color_blending_create_info.sType =
00479       VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
00480   color_blending_create_info.logicOpEnable = VK_FALSE;
00481   color_blending_create_info.attachmentCount = 1;
00482   color_blending_create_info.pAttachments = &color_state;
00483
00484   // -- PIPELINE LAYOUT --
00485   VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00486   pipeline_layout_create_info.sType =
00487       VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00488   pipeline_layout_create_info.setLayoutCount =
00489       static_cast<uint32_t>(descriptorSetLayouts.size());
00490   pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00491   pipeline_layout_create_info.pushConstantRangeCount = 1;
00492   pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;
00493
00494   // create pipeline layout
00495   VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00496                                            &pipeline_layout_create_info,
00497                                            nullptr, &pipeline_layout);
00498   ASSERT_VULKAN(result, "Failed to create pipeline layout!")
00499
00500   // -- DEPTH STENCIL TESTING --
00501   VkPipelineDepthStencilStateCreateInfo depth_stencil_create_info{};
00502   depth_stencil_create_info.sType =
00503       VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
00504   depth_stencil_create_info.depthTestEnable = VK_TRUE;
00505   depth_stencil_create_info.depthWriteEnable = VK_TRUE;
```

```
00506   depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS;
00507   depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
00508   depth_stencil_create_info.stencilTestEnable = VK_FALSE;
00509
00510   // -- GRAPHICS PIPELINE CREATION --
00511   VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
00512   graphics_pipeline_create_info.sType =
00513       VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
00514   graphics_pipeline_create_info.stageCount =
00515       static_cast<uint32_t>(shader_stages.size());
00516   graphics_pipeline_create_info.pStages = shader_stages.data();
00517   graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
00518   graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
00519   graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
00520   graphics_pipeline_create_info.pDynamicState = nullptr;
00521   graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
00522   graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
00523   graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
00524   graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
00525   graphics_pipeline_create_info.layout = pipeline_layout;
00526   graphics_pipeline_create_info.renderPass = render_pass;
00527   graphics_pipeline_create_info.subpass = 0;
00528
00529   // pipeline derivatives : can create multiple pipelines that derive from one
00530   // another for optimization
00531   graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
00532   graphics_pipeline_create_info.basePipelineIndex = -1;
00533
00534   // create graphics pipeline
00535   result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
00536                                      1, &graphics_pipeline_create_info, nullptr,
00537                                      &graphics_pipeline);
00538   ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")
00539
00540   // Destroy shader modules, no longer needed after pipeline created
00541   vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
00542                         nullptr);
00543   vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,
00544                         nullptr);
00545 }
```

4.19 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/renderer/Raytracing.cpp File Reference

```
#include "Raytracing.h"
#include <array>
#include <vector>
#include "File.h"
#include "MemoryHelper.h"
#include "ShaderHelper.h"
```

Include dependency graph for Raytracing.cpp:

4.20 Raytracing.cpp

Go to the documentation of this file.

```
00001 #include "Raytracing.h"
00002
00003 #include <array>
00004 #include <vector>
00005
00006 #include "File.h"
00007 #include "MemoryHelper.h"
00008 #include "ShaderHelper.h"
00009
00010 Raytracing::Raytracing() {}
00011
00012 void Raytracing::init(
00013     VulkanDevice* device,
00014     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00015   this->device = device;
00016
```

```
00017   createPCRange();
00018   createGraphicsPipeline(descriptorSetLayouts);
00019   createSBT();
00020 }
00021
00022 void Raytracing::shaderHotReload(
00023     const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
00024   vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
00025   createGraphicsPipeline(descriptor_set_layouts);
00026 }
00027
00028 void Raytracing::recordCommands(
00029     VkCommandBuffer& commandBuffer, VulkanSwapChain* vulkanSwapChain,
00030     const std::vector<VkDescriptorSet>& descriptorSets) {
00031   uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
00032   uint32_t handle_size_aligned =
00033       align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);
00034
00035   PFN_vkGetBufferDeviceAddressKHR vkGetBufferDeviceAddressKHR =
00036       reinterpret_cast<PFN_vkGetBufferDeviceAddressKHR>(vkGetDeviceProcAddr(
00037           device->getLogicalDevice(), "vkGetBufferDeviceAddressKHR"));
00038
00039   PFN_vkCmdTraceRaysKHR pvkCmdTraceRaysKHR =
00040       (PFN_vkCmdTraceRaysKHR)vkGetDeviceProcAddr(device->getLogicalDevice(),
00041                                                  "vkCmdTraceRaysKHR");
00042
00043   VkBufferDeviceAddressInfoKHR bufferDeviceAI{};
00044   bufferDeviceAI.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
00045   bufferDeviceAI.buffer = raygenShaderBindingTableBuffer.getBuffer();
00046
00047   rgen_region.deviceAddress =
00048       vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00049   rgen_region.stride = handle_size_aligned;
00050   rgen_region.size = handle_size_aligned;
00051
00052   bufferDeviceAI.buffer = missShaderBindingTableBuffer.getBuffer();
00053   miss_region.deviceAddress =
00054       vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00055   miss_region.stride = handle_size_aligned;
00056   miss_region.size = handle_size_aligned;
00057
00058   bufferDeviceAI.buffer = hitShaderBindingTableBuffer.getBuffer();
00059   hit_region.deviceAddress =
00060       vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
00061   hit_region.stride = handle_size_aligned;
00062   hit_region.size = handle_size_aligned;
00063
00064   // for GCC doen't allow references on rvalues go like that ...
00065   pc.clear_color = {0.2f, 0.65f, 0.4f, 1.0f};
00066   // just "Push" constants to given shader stage directly (no buffer)
00067   vkCmdPushConstants(commandBuffer, pipeline_layout,
00068                      VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00069                          VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00070                          VK_SHADER_STAGE_MISS_BIT_KHR,
00071                      0, sizeof(PushConstantRaytracing), &pc);
00072
00073   vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
00074                     graphicsPipeline);
00075
00076   vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
00077                           pipeline_layout, 0,
00078                           static_cast<uint32_t>(descriptorSets.size()),
00079                           descriptorSets.data(), 0, nullptr);
00080
00081   const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
00082   pvkCmdTraceRaysKHR(commandBuffer, &rgen_region, &miss_region, &hit_region,
00083                      &call_region, swap_chain_extent.width,
00084                      swap_chain_extent.height, 1);
00085 }
00086
00087 void Raytracing::cleanUp() {
00088   shaderBindingTableBuffer.cleanUp();
00089   raygenShaderBindingTableBuffer.cleanUp();
00090   missShaderBindingTableBuffer.cleanUp();
00091   hitShaderBindingTableBuffer.cleanUp();
00092
00093   vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
00094   vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
00095 }
00096
00097 Raytracing::~Raytracing() {}
00098
00099 void Raytracing::createPCRange() {
00100   // define push constant values (no 'create' needed)
00101   pc_ranges.stageFlags = VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00102                          VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00103                          VK_SHADER_STAGE_MISS_BIT_KHR;
```

```
00104   pc_ranges.offset = 0;
00105   pc_ranges.size = sizeof(PushConstantRaytracing);  // size of data being passed
00106 }
00107
00108 void Raytracing::createGraphicsPipeline(
00109     const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
00110   PFN_vkCreateRayTracingPipelinesKHR pvkCreateRayTracingPipelinesKHR =
00111       (PFN_vkCreateRayTracingPipelinesKHR)vkGetDeviceProcAddr(
00112           device->getLogicalDevice(), "vkCreateRayTracingPipelinesKHR");
00113
00114   std::stringstream raytracing_shader_dir;
00115   raytracing_shader_dir « CMAKELISTS_DIR;
00116   raytracing_shader_dir « "/Resources/Shader/raytracing/";
00117
00118   std::string raygen_shader = "raytrace.rgen";
00119   std::string chit_shader = "raytrace.rchit";
00120   std::string miss_shader = "raytrace.rmiss";
00121   std::string shadow_shader = "shadow.rmiss";
00122
00123   ShaderHelper shaderHelper;
00124   shaderHelper.compileShader(raytracing_shader_dir.str(), raygen_shader);
00125   shaderHelper.compileShader(raytracing_shader_dir.str(), chit_shader);
00126   shaderHelper.compileShader(raytracing_shader_dir.str(), miss_shader);
00127   shaderHelper.compileShader(raytracing_shader_dir.str(), shadow_shader);
00128
00129   File raygenFile(
00130       shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), raygen_shader));
00131   File raychitFile(
00132       shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), chit_shader));
00133   File raymissFile(
00134       shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), miss_shader));
00135   File shadowFile(
00136       shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), shadow_shader));
00137
00138   std::vector<char> raygen_shader_code = raygenFile.readCharSequence();
00139   std::vector<char> raychit_shader_code = raychitFile.readCharSequence();
00140   std::vector<char> raymiss_shader_code = raymissFile.readCharSequence();
00141   std::vector<char> shadow_shader_code = shadowFile.readCharSequence();
00142
00143   // build shader modules to link to graphics pipeline
00144   VkShaderModule raygen_shader_module =
00145       shaderHelper.createShaderModule(device, raygen_shader_code);
00146   VkShaderModule raychit_shader_module =
00147       shaderHelper.createShaderModule(device, raychit_shader_code);
00148   VkShaderModule raymiss_shader_module =
00149       shaderHelper.createShaderModule(device, raymiss_shader_code);
00150   VkShaderModule shadow_shader_module =
00151       shaderHelper.createShaderModule(device, shadow_shader_code);
00152
00153   // create all shader stage infos for creating a group
00154   VkPipelineShaderStageCreateInfo rgen_shader_stage_info{};
00155   rgen_shader_stage_info.sType =
00156       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00157   rgen_shader_stage_info.stage = VK_SHADER_STAGE_RAYGEN_BIT_KHR;
00158   rgen_shader_stage_info.module = raygen_shader_module;
00159   rgen_shader_stage_info.pName = "main";
00160
00161   VkPipelineShaderStageCreateInfo rmiss_shader_stage_info{};
00162   rmiss_shader_stage_info.sType =
00163       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00164   rmiss_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
00165   rmiss_shader_stage_info.module = raymiss_shader_module;
00166   rmiss_shader_stage_info.pName = "main";
00167
00168   VkPipelineShaderStageCreateInfo shadow_shader_stage_info{};
00169   shadow_shader_stage_info.sType =
00170       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00171   shadow_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
00172   shadow_shader_stage_info.module = shadow_shader_module;
00173   shadow_shader_stage_info.pName = "main";
00174
00175   VkPipelineShaderStageCreateInfo rchit_shader_stage_info{};
00176   rchit_shader_stage_info.sType =
00177       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
00178   rchit_shader_stage_info.stage = VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR;
00179   rchit_shader_stage_info.module = raychit_shader_module;
00180   rchit_shader_stage_info.pName = "main";
00181
00182   // we have all shader stages together
00183   std::array<VkPipelineShaderStageCreateInfo, 4> shader_stages = {
00184       rgen_shader_stage_info, rmiss_shader_stage_info, shadow_shader_stage_info,
00185       rchit_shader_stage_info};
00186
00187   enum StageIndices { eRaygen, eMiss, eMiss2, eClosestHit, eShaderGroupCount };
00188
00189   shader_groups.reserve(4);
00190   VkRayTracingShaderGroupCreateInfoKHR shader_group_create_infos[4];
```

```
00191
00192   shader_group_create_infos[0].sType =
00193       VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00194   shader_group_create_infos[0].pNext = nullptr;
00195   shader_group_create_infos[0].type =
00196       VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00197   shader_group_create_infos[0].generalShader = eRaygen;
00198   shader_group_create_infos[0].closestHitShader = VK_SHADER_UNUSED_KHR;
00199   shader_group_create_infos[0].anyHitShader = VK_SHADER_UNUSED_KHR;
00200   shader_group_create_infos[0].intersectionShader = VK_SHADER_UNUSED_KHR;
00201   shader_group_create_infos[0].pShaderGroupCaptureReplayHandle = nullptr;
00202
00203   shader_groups.push_back(shader_group_create_infos[0]);
00204
00205   shader_group_create_infos[1].sType =
00206       VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00207   shader_group_create_infos[1].pNext = nullptr;
00208   shader_group_create_infos[1].type =
00209       VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00210   shader_group_create_infos[1].generalShader = eMiss;
00211   shader_group_create_infos[1].closestHitShader = VK_SHADER_UNUSED_KHR;
00212   shader_group_create_infos[1].anyHitShader = VK_SHADER_UNUSED_KHR;
00213   shader_group_create_infos[1].intersectionShader = VK_SHADER_UNUSED_KHR;
00214   shader_group_create_infos[1].pShaderGroupCaptureReplayHandle = nullptr;
00215
00216   shader_groups.push_back(shader_group_create_infos[1]);
00217
00218   shader_group_create_infos[2].sType =
00219       VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00220   shader_group_create_infos[2].pNext = nullptr;
00221   shader_group_create_infos[2].type =
00222       VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
00223   shader_group_create_infos[2].generalShader = eMiss2;
00224   shader_group_create_infos[2].closestHitShader = VK_SHADER_UNUSED_KHR;
00225   shader_group_create_infos[2].anyHitShader = VK_SHADER_UNUSED_KHR;
00226   shader_group_create_infos[2].intersectionShader = VK_SHADER_UNUSED_KHR;
00227   shader_group_create_infos[2].pShaderGroupCaptureReplayHandle = nullptr;
00228
00229   shader_groups.push_back(shader_group_create_infos[2]);
00230
00231   shader_group_create_infos[3].sType =
00232       VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
00233   shader_group_create_infos[3].pNext = nullptr;
00234   shader_group_create_infos[3].type =
00235       VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR;
00236   shader_group_create_infos[3].generalShader = VK_SHADER_UNUSED_KHR;
00237   shader_group_create_infos[3].closestHitShader = eClosestHit;
00238   shader_group_create_infos[3].anyHitShader = VK_SHADER_UNUSED_KHR;
00239   shader_group_create_infos[3].intersectionShader = VK_SHADER_UNUSED_KHR;
00240   shader_group_create_infos[3].pShaderGroupCaptureReplayHandle = nullptr;
00241
00242   shader_groups.push_back(shader_group_create_infos[3]);
00243
00244   VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
00245   pipeline_layout_create_info.sType =
00246       VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
00247   pipeline_layout_create_info.setLayoutCount =
00248       static_cast<uint32_t>(descriptorSetLayouts.size());
00249   pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
00250   pipeline_layout_create_info.pushConstantRangeCount = 1;
00251   pipeline_layout_create_info.pPushConstantRanges = &pc_ranges;
00252
00253   VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
00254                                            &pipeline_layout_create_info,
00255                                            nullptr, &pipeline_layout);
00256   ASSERT_VULKAN(result, "Failed to create raytracing pipeline layout!")
00257
00258   VkPipelineLibraryCreateInfoKHR pipeline_library_create_info{};
00259   pipeline_library_create_info.sType =
00260       VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR;
00261   pipeline_library_create_info.pNext = nullptr;
00262   pipeline_library_create_info.libraryCount = 0;
00263   pipeline_library_create_info.pLibraries = nullptr;
00264
00265   VkRayTracingPipelineCreateInfoKHR raytracing_pipeline_create_info{};
00266   raytracing_pipeline_create_info.sType =
00267       VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR;
00268   raytracing_pipeline_create_info.pNext = nullptr;
00269   raytracing_pipeline_create_info.flags = 0;
00270   raytracing_pipeline_create_info.stageCount =
00271       static_cast<uint32_t>(shader_stages.size());
00272   raytracing_pipeline_create_info.pStages = shader_stages.data();
00273   raytracing_pipeline_create_info.groupCount =
00274       static_cast<uint32_t>(shader_groups.size());
00275   raytracing_pipeline_create_info.pGroups = shader_groups.data();
00276   /*raytracing_pipeline_create_info.pLibraryInfo =
00277      &pipeline_library_create_info;
```

```
00278         raytracing_pipeline_create_info.pLibraryInterface = NULL;*/
00279   // TODO: HARDCODED FOR NOW;
00280   raytracing_pipeline_create_info.maxPipelineRayRecursionDepth = 2;
00281   raytracing_pipeline_create_info.layout = pipeline_layout;
00282
00283   result = pvkCreateRayTracingPipelinesKHR(
00284       device->getLogicalDevice(), VK_NULL_HANDLE, VK_NULL_HANDLE, 1,
00285       &raytracing_pipeline_create_info, nullptr, &graphicsPipeline);
00286
00287   ASSERT_VULKAN(result, "Failed to create raytracing pipeline!")
00288
00289   vkDestroyShaderModule(device->getLogicalDevice(), raygen_shader_module,
00290                         nullptr);
00291   vkDestroyShaderModule(device->getLogicalDevice(), raymiss_shader_module,
00292                         nullptr);
00293   vkDestroyShaderModule(device->getLogicalDevice(), raychit_shader_module,
00294                         nullptr);
00295   vkDestroyShaderModule(device->getLogicalDevice(), shadow_shader_module,
00296                         nullptr);
00297 }
00298
00299 void Raytracing::createSBT() {
00300   // load in functionality for raytracing shader group handles
00301   PFN_vkGetRayTracingShaderGroupHandlesKHR
00302       pvkGetRayTracingShaderGroupHandlesKHR =
00303           (PFN_vkGetRayTracingShaderGroupHandlesKHR)vkGetDeviceProcAddr(
00304               device->getLogicalDevice(),
00305               "vkGetRayTracingShaderGroupHandlesKHR");
00306
00307   raytracing_properties = VkPhysicalDeviceRayTracingPipelinePropertiesKHR{};
00308   raytracing_properties.sType =
00309       VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR;
00310
00311   VkPhysicalDeviceProperties2 properties{};
00312   properties.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;
00313   properties.pNext = &raytracing_properties;
00314
00315   vkGetPhysicalDeviceProperties2(device->getPhysicalDevice(), &properties);
00316
00317   uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
00318   uint32_t handle_size_aligned =
00319       align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);
00320
00321   uint32_t group_count = static_cast<uint32_t>(shader_groups.size());
00322   uint32_t sbt_size = group_count * handle_size_aligned;
00323
00324   std::vector<uint8_t> handles(sbt_size);
00325
00326   VkResult result = pvkGetRayTracingShaderGroupHandlesKHR(
00327       device->getLogicalDevice(), graphicsPipeline, 0, group_count, sbt_size,
00328       handles.data());
00329   ASSERT_VULKAN(result, "Failed to get ray tracing shader group handles!")
00330
00331   const VkBufferUsageFlags bufferUsageFlags =
00332       VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR |
00333       VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT;
00334   const VkMemoryPropertyFlags memoryUsageFlags =
00335       VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
00336       VK_MEMORY_PROPERTY_HOST_COHERENT_BIT;
00337
00338   raygenShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
00339                                         memoryUsageFlags);
00340
00341   missShaderBindingTableBuffer.create(device, 2 * handle_size, bufferUsageFlags,
00342                                       memoryUsageFlags);
00343
00344   hitShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
00345                                      memoryUsageFlags);
00346
00347   void* mapped_raygen = nullptr;
00348   vkMapMemory(device->getLogicalDevice(),
00349               raygenShaderBindingTableBuffer.getBufferMemory(), 0,
00350               VK_WHOLE_SIZE, 0, &mapped_raygen);
00351
00352   void* mapped_miss = nullptr;
00353   vkMapMemory(device->getLogicalDevice(),
00354               missShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
00355               0, &mapped_miss);
00356
00357   void* mapped_rchit = nullptr;
00358   vkMapMemory(device->getLogicalDevice(),
00359               hitShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
00360               0, &mapped_rchit);
00361
00362   memcpy(mapped_raygen, handles.data(), handle_size);
00363   memcpy(mapped_miss, handles.data() + handle_size_aligned, handle_size * 2);
00364   memcpy(mapped_rchit, handles.data() + handle_size_aligned * 3, handle_size);
```

```
00365 }
```

4.21 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/renderer/↩ VulkanRenderer.cpp File Reference

```
#include "VulkanRenderer.hpp"
#include <algorithm>
#include <iostream>
#include <vector>
#include <vk_mem_alloc.h>
#include <stb_image.h>
#include <gsl/gsl>
#include "File.h"
#include "Globals.h"
#include "PushConstantPost.h"
#include "ShaderHelper.h"
```

Include dependency graph for VulkanRenderer.cpp:

Macros

- #define VMA_IMPLEMENTATION
- #define STB_IMAGE_IMPLEMENTATION

4.21.1 Macro Definition Documentation

4.21.1.1 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

Definition at line 12 of file VulkanRenderer.cpp.

4.21.1.2 VMA_IMPLEMENTATION

```
#define VMA_IMPLEMENTATION
```

Definition at line 8 of file VulkanRenderer.cpp.

4.22 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
00010 #include <vk_mem_alloc.h>
00011
00012 #define STB_IMAGE_IMPLEMENTATION
00013 #include <stb_image.h>
00014
00015 #include <gsl/gsl>
00016
00017 #include "File.h"
00018 #include "Globals.h"
00019 #include "PushConstantPost.h"
00020 #include "ShaderHelper.h"
00021
00022 VulkanRenderer::VulkanRenderer(Window* window, Scene* scene, GUI* gui,
00023                                Camera* camera)
00024     :
00025
00026       window(window),
00027       scene(scene),
00028       gui(gui)
00029
00030 {
00031   updateUniforms(scene, camera, window);
00032
00033   try {
00034     instance = VulkanInstance();
00035
00036     VkDebugReportFlagsEXT debugReportFlags =
00037         VK_DEBUG_REPORT_ERROR_BIT_EXT | VK_DEBUG_REPORT_WARNING_BIT_EXT;
00038     if (ENABLE_VALIDATION_LAYERS)
00039       debug::setupDebugging(instance.getVulkanInstance(), debugReportFlags,
00040                             VK_NULL_HANDLE);
00041
00042     create_surface();
00043
00044     device = std::make_unique<VulkanDevice>(&instance, &surface);
00045
00046     allocator =
00047         Allocator(device->getLogicalDevice(), device->getPhysicalDevice(),
00048                   instance.getVulkanInstance());
00049
00050     create_command_pool();
00051
00052     vulkanSwapChain.initVulkanContext(device.get(), window, surface);
00053     create_uniform_buffers();
00054     create_command_buffers();
00055
00056     createSynchronization();
00057
00058     createSharedRenderDescriptorSetLayouts();
00059     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_rasterizer = {
00060         sharedRenderDescriptorSetLayout};
00061     rasterizer.init(device.get(), &vulkanSwapChain,
00062                     descriptor_set_layouts_rasterizer, graphics_command_pool);
00063     create_post_descriptor_layout();
00064     std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
00065         post_descriptor_set_layout};
00066     postStage.init(device.get(), &vulkanSwapChain, descriptor_set_layouts_post);
00067     createDescriptorPoolSharedRenderStages();
00068     createSharedRenderDescriptorSet();
00069
00070     updatePostDescriptorSets();
00071
00072     createRaytracingDescriptorPool();
00073     createRaytracingDescriptorSetLayouts();
00074     std::vector<VkDescriptorSetLayout> layouts;
00075     layouts.push_back(sharedRenderDescriptorSetLayout);
00076     layouts.push_back(raytracingDescriptorSetLayout);
00077     raytracingStage.init(device.get(), layouts);
00078     pathTracing.init(device.get(), layouts);
00079
00080     scene->loadModel(device.get(), graphics_command_pool);
00081     updateTexturesInSharedRenderDescriptorSet();
00082
```

```
00083     asManager.createASForScene(device.get(), graphics_command_pool, scene);
00084     create_object_description_buffer();
00085     createRaytracingDescriptorSets();
00086     updateRaytracingDescriptorSets();
00087
00088     gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
00089                                  postStage.getRenderPass(),
00090                                  graphics_command_pool);
00091
00092   } catch (const std::runtime_error& e) {
00093     printf("ERROR: %s\n", e.what());
00094   }
00095 }
00096
00097 void VulkanRenderer::updateUniforms(Scene* scene, Camera* camera,
00098                                     Window* window) {
00099   const GUISceneSharedVars guiSceneSharedVars = scene->getGuiSceneSharedVars();
00100
00101   globalUBO.view = camera->calculate_viewmatrix();
00102   globalUBO.projection =
00103       glm::perspective(glm::radians(camera->get_fov()),
00104                        (float)window->get_width() / (float)window->get_height(),
00105                        camera->get_near_plane(), camera->get_far_plane());
00106
00107   sceneUBO.view_dir = glm::vec4(camera->get_camera_direction(), 1.0f);
00108
00109   sceneUBO.light_dir =
00110       glm::vec4(guiSceneSharedVars.directional_light_direction[0],
00111                 guiSceneSharedVars.directional_light_direction[1],
00112                 guiSceneSharedVars.directional_light_direction[2], 1.0f);
00113
00114   sceneUBO.cam_pos =
00115       glm::vec4(camera->get_camera_position(), camera->get_fov());
00116 }
00117
00118 void VulkanRenderer::updateStateDueToUserInput(GUI* gui) {
00119   GUIRendererSharedVars& guiRendererSharedVars =
00120       gui->getGuiRendererSharedVars();
00121
00122   if (guiRendererSharedVars.shader_hot_reload_triggered) {
00123     shaderHotReload();
00124     guiRendererSharedVars.shader_hot_reload_triggered = false;
00125   }
00126 }
00127
00128 void VulkanRenderer::finishAllRenderCommands() {
00129   vkDeviceWaitIdle(device->getLogicalDevice());
00130 }
00131
00132 void VulkanRenderer::shaderHotReload() {
00133   // wait until no actions being run on device before destroying
00134   vkDeviceWaitIdle(device->getLogicalDevice());
00135
00136   std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
00137       sharedRenderDescriptorSetLayout};
00138   rasterizer.shaderHotReload(descriptor_set_layouts);
00139
00140   std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
00141       post_descriptor_set_layout};
00142   postStage.shaderHotReload(descriptor_set_layouts_post);
00143
00144   std::vector<VkDescriptorSetLayout> layouts = {sharedRenderDescriptorSetLayout,
00145                                                 raytracingDescriptorSetLayout};
00146   raytracingStage.shaderHotReload(layouts);
00147   pathTracing.shaderHotReload(layouts);
00148 }
00149
00150 void VulkanRenderer::drawFrame() {
00151   // We need to skip one frame
00152   // Due to ImGui need to call ImGui::NewFrame() again
00153   // if we recreated swapchain
00154   if (checkChangedFramebufferSize()) return;
00155
00156   /*1. Get next available image to draw to and set something to signal when
00157      we're finished with the image  (a semaphore) wait for given fence to signal
00158      (open) from last draw before continuing*/
00159   VkResult result = vkWaitForFences(device->getLogicalDevice(), 1,
00160                                     &in_flight_fences[current_frame], VK_TRUE,
00161                                     std::numeric_limits<uint64_t>::max());
00162   ASSERT_VULKAN(result, "Failed to wait for fences!")
00163   // -- GET NEXT IMAGE --
00164   uint32_t image_index;
00165   result = vkAcquireNextImageKHR(
00166       device->getLogicalDevice(), vulkanSwapChain.getSwapChain(),
00167       std::numeric_limits<uint64_t>::max(), image_available[current_frame],
00168       VK_NULL_HANDLE, &image_index);
00169
```

```
00170   if (result == VK_ERROR_OUT_OF_DATE_KHR) {
00171     // recreate_swap_chain();
00172     return;
00173
00174   } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
00175     throw std::runtime_error("Failed to acquire next image!");
00176   }
00177
00178   //// check if previous frame is using this image (i.e. there is its fence to
00179   /// wait on)
00180   if (images_in_flight_fences[image_index] != VK_NULL_HANDLE) {
00181     vkWaitForFences(device->getLogicalDevice(), 1,
00182                     &images_in_flight_fences[image_index], VK_TRUE, UINT64_MAX);
00183   }
00184
00185   // mark the image as now being in use by this frame
00186   images_in_flight_fences[image_index] = in_flight_fences[current_frame];
00187
00188   VkCommandBufferBeginInfo buffer_begin_info{};
00189   buffer_begin_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
00190   buffer_begin_info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
00191   // start recording commands to command buffer
00192   result =
00193       vkBeginCommandBuffer(command_buffers[image_index], &buffer_begin_info);
00194   ASSERT_VULKAN(result, "Failed to start recording a command buffer!")
00195
00196   update_uniform_buffers(image_index);
00197
00198   GUIRendererSharedVars& guiRendererSharedVars =
00199       gui->getGuiRendererSharedVars();
00200   if (guiRendererSharedVars.raytracing)
00201     update_raytracing_descriptor_set(image_index);
00202
00203   record_commands(image_index);
00204
00205   // stop recording to command buffer
00206   result = vkEndCommandBuffer(command_buffers[image_index]);
00207   ASSERT_VULKAN(result, "Failed to stop recording a command buffer!")
00208
00209   // 2. Submit command buffer to queue for execution, making sure it waits for
00210   // the image to be signalled as available before drawing and signals when it
00211   // has finished rendering
00212   // -- SUBMIT COMMAND BUFFER TO RENDER --
00213   VkSubmitInfo submit_info{};
00214   submit_info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
00215   submit_info.waitSemaphoreCount = 1;  // number of semaphores to wait on
00216   submit_info.pWaitSemaphores =
00217       &image_available[current_frame];  // list of semaphores to wait on
00218
00219   VkPipelineStageFlags wait_stages = {
00220
00221       VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT /*|
00222                   VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT |
00223                   VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR*/
00224
00225   };
00226
00227   submit_info.pWaitDstStageMask =
00228       &wait_stages;  // stages to check semaphores at
00229
00230   submit_info.commandBufferCount = 1;  // number of command buffers to submit
00231   submit_info.pCommandBuffers =
00232       &command_buffers[image_index];     // command buffer to submit
00233   submit_info.signalSemaphoreCount = 1;  // number of semaphores to signal
00234   submit_info.pSignalSemaphores =
00235       &render_finished[current_frame];  // semaphores to signal when command
00236                                         // buffer finishes
00237
00238   result = vkResetFences(device->getLogicalDevice(), 1,
00239                          &in_flight_fences[current_frame]);
00240   ASSERT_VULKAN(result, "Failed to reset fences!")
00241
00242   // submit command buffer to queue
00243   result = vkQueueSubmit(device->getGraphicsQueue(), 1, &submit_info,
00244                          in_flight_fences[current_frame]);
00245   ASSERT_VULKAN(result, "Failed to submit command buffer to queue!")
00246
00247   // 3. Present image to screen when it has signalled finished rendering
00248   // -- PRESENT RENDERED IMAGE TO SCREEN --
00249   VkPresentInfoKHR present_info{};
00250   present_info.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
00251   present_info.waitSemaphoreCount = 1;  // number of semaphores to wait on
00252   present_info.pWaitSemaphores =
00253       &render_finished[current_frame];  // semaphores to wait on
00254   present_info.swapchainCount = 1;      // number of swapchains to present to
00255   const VkSwapchainKHR swapchain = vulkanSwapChain.getSwapChain();
00256   present_info.pSwapchains = &swapchain;  // swapchains to present images to
```

```
00257   present_info.pImageIndices =
00258       &image_index;  // index of images in swapchain to present
00259
00260   result = vkQueuePresentKHR(device->getPresentationQueue(), &present_info);
00261
00262   if (result == VK_ERROR_OUT_OF_DATE_KHR) {
00263     // recreate_swap_chain();
00264     return;
00265
00266   } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
00267     throw std::runtime_error("Failed to acquire next image!");
00268   }
00269
00270   if (result != VK_SUCCESS) {
00271     throw std::runtime_error("Failed to submit to present queue!");
00272   }
00273
00274   current_frame = (current_frame + 1) % MAX_FRAME_DRAWS;
00275 }
00276
00277 void VulkanRenderer::create_surface() {
00278   // create surface (creates a surface create info struct, runs the create
00279   // surface function, returns result)
00280   ASSERT_VULKAN(
00281       glfwCreateWindowSurface(instance.getVulkanInstance(),
00282                               window->get_window(), nullptr, &surface),
00283       "Failed to create a surface!");
00284 }
00285
00286 void VulkanRenderer::create_post_descriptor_layout() {
00287   // UNIFORM VALUES DESCRIPTOR SET LAYOUT
00288   // globalUBO Binding info
00289   VkDescriptorSetLayoutBinding post_sampler_layout_binding{};
00290   post_sampler_layout_binding.binding =
00291       0;  // binding point in shader (designated by binding number in shader)
00292   post_sampler_layout_binding.descriptorType =
00293       VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;  // type of descriptor
00294                                                   // (uniform, dynamic uniform,
00295                                                   // image sampler, etc)
00296   post_sampler_layout_binding.descriptorCount =
00297       1;  // number of descriptors for binding
00298   post_sampler_layout_binding.stageFlags =
00299       VK_SHADER_STAGE_FRAGMENT_BIT;  // we need to say at which shader we bind
00300                                      // this uniform to
00301   post_sampler_layout_binding.pImmutableSamplers =
00302       nullptr;  // for texture: can make sampler data unchangeable (immutable)
00303                 // by specifying in layout
00304
00305   std::vector<VkDescriptorSetLayoutBinding> layout_bindings = {
00306       post_sampler_layout_binding};
00307
00308   // create descriptor set layout with given bindings
00309   VkDescriptorSetLayoutCreateInfo layout_create_info{};
00310   layout_create_info.sType =
00311       VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00312   layout_create_info.bindingCount = static_cast<uint32_t>(
00313       layout_bindings.size());  // only have 1 for the globalUBO
00314   layout_create_info.pBindings =
00315       layout_bindings.data();  // array of binding infos
00316
00317   // create descriptor set layout
00318   VkResult result = vkCreateDescriptorSetLayout(device->getLogicalDevice(),
00319                                                 &layout_create_info, nullptr,
00320                                                 &post_descriptor_set_layout);
00321   ASSERT_VULKAN(result, "Failed to create descriptor set layout!")
00322
00323   VkDescriptorPoolSize post_pool_size{};
00324   post_pool_size.type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
00325   post_pool_size.descriptorCount = static_cast<uint32_t>(1);
00326
00327   // list of pool sizes
00328   std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {post_pool_size};
00329
00330   VkDescriptorPoolCreateInfo pool_create_info{};
00331   pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00332   pool_create_info.maxSets =
00333       vulkanSwapChain
00334           .getNumberSwapChainImages();  // maximum number of descriptor sets
00335                                         // that can be created from pool
00336   pool_create_info.poolSizeCount = static_cast<uint32_t>(
00337       descriptor_pool_sizes.size());  // amount of pool sizes being passed
00338   pool_create_info.pPoolSizes =
00339       descriptor_pool_sizes.data();  // pool sizes to create pool with
00340
00341   // create descriptor pool
00342   result = vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
00343                                   nullptr, &post_descriptor_pool);
```

```
00344   ASSERT_VULKAN(result, "Failed to create a descriptor pool!")
00345
00346   // resize descriptor set list so one for every buffer
00347   post_descriptor_set.resize(vulkanSwapChain.getNumberSwapChainImages());
00348
00349   std::vector<VkDescriptorSetLayout> set_layouts(
00350       vulkanSwapChain.getNumberSwapChainImages(), post_descriptor_set_layout);
00351
00352   // descriptor set allocation info
00353   VkDescriptorSetAllocateInfo set_alloc_info{};
00354   set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00355   set_alloc_info.descriptorPool =
00356       post_descriptor_pool;  // pool to allocate descriptor set from
00357   set_alloc_info.descriptorSetCount =
00358       vulkanSwapChain.getNumberSwapChainImages();  // number of sets to allocate
00359   set_alloc_info.pSetLayouts =
00360       set_layouts.data();  // layouts to use to allocate sets (1:1 relationship)
00361
00362   // allocate descriptor sets (multiple)
00363   result = vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
00364                                     post_descriptor_set.data());
00365   ASSERT_VULKAN(result, "Failed to create descriptor sets!")
00366 }
00367
00368 void VulkanRenderer::updatePostDescriptorSets() {
00369   // update all of descriptor set buffer bindings
00370   for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00371     // texture image info
00372     VkDescriptorImageInfo image_info{};
00373     image_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00374     Texture& renderResult = rasterizer.getOffscreenTexture(i);
00375     image_info.imageView = renderResult.getImageView();
00376     image_info.sampler = postStage.getOffscreenSampler();
00377
00378     // descriptor write info
00379     VkWriteDescriptorSet descriptor_write{};
00380     descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00381     descriptor_write.dstSet = post_descriptor_set[i];
00382     descriptor_write.dstBinding = 0;
00383     descriptor_write.dstArrayElement = 0;
00384     descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
00385     descriptor_write.descriptorCount = 1;
00386     descriptor_write.pImageInfo = &image_info;
00387
00388     // update new descriptor set
00389     vkUpdateDescriptorSets(device->getLogicalDevice(), 1, &descriptor_write, 0,
00390                            nullptr);
00391   }
00392 }
00393
00394 void VulkanRenderer::createRaytracingDescriptorPool() {
00395   std::array<VkDescriptorPoolSize, 2> descriptor_pool_sizes{};
00396
00397   descriptor_pool_sizes[0].type = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00398   descriptor_pool_sizes[0].descriptorCount = 1;
00399
00400   descriptor_pool_sizes[1].type = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00401   descriptor_pool_sizes[1].descriptorCount = 1;
00402
00403   VkDescriptorPoolCreateInfo descriptor_pool_create_info{};
00404   descriptor_pool_create_info.sType =
00405       VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00406   descriptor_pool_create_info.poolSizeCount =
00407       static_cast<uint32_t>(descriptor_pool_sizes.size());
00408   descriptor_pool_create_info.pPoolSizes = descriptor_pool_sizes.data();
00409   descriptor_pool_create_info.maxSets =
00410       vulkanSwapChain.getNumberSwapChainImages();
00411
00412   VkResult result = vkCreateDescriptorPool(device->getLogicalDevice(),
00413                                            &descriptor_pool_create_info,
00414                                            nullptr, &raytracingDescriptorPool);
00415   ASSERT_VULKAN(result, "Failed to create command pool!")
00416 }
00417
00418 void VulkanRenderer::cleanUpSync() {
00419   for (int i = 0; i < MAX_FRAME_DRAWS; i++) {
00420     vkDestroySemaphore(device->getLogicalDevice(), render_finished[i], nullptr);
00421     vkDestroySemaphore(device->getLogicalDevice(), image_available[i], nullptr);
00422     vkDestroyFence(device->getLogicalDevice(), in_flight_fences[i], nullptr);
00423   }
00424 }
00425
00426 void VulkanRenderer::create_object_description_buffer() {
00427   std::vector<ObjectDescription> objectDescriptions =
00428       scene->getObjectDescriptions();
00429
00430   vulkanBufferManager.createBufferAndUploadVectorOnDevice(
```

```
00431       device.get(), graphics_command_pool, objectDescriptionBuffer,
00432       VK_BUFFER_USAGE_TRANSFER_DST_BIT |
00433           VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
00434           VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
00435       VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
00436           VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
00437       objectDescriptions);
00438
00439   // update the object description set
00440   // update all of descriptor set buffer bindings
00441   for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00442     VkDescriptorBufferInfo object_descriptions_buffer_info{};
00443     // image_info.sampler = VK_DESCRIPTOR_TYPE_SAMPLER;
00444     object_descriptions_buffer_info.buffer =
00445         objectDescriptionBuffer.getBuffer();
00446     object_descriptions_buffer_info.offset = 0;
00447     object_descriptions_buffer_info.range = VK_WHOLE_SIZE;
00448
00449     VkWriteDescriptorSet descriptor_object_descriptions_writer{};
00450     descriptor_object_descriptions_writer.sType =
00451         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00452     descriptor_object_descriptions_writer.pNext = nullptr;
00453     descriptor_object_descriptions_writer.dstSet = sharedRenderDescriptorSet[i];
00454     descriptor_object_descriptions_writer.dstBinding =
00455         OBJECT_DESCRIPTION_BINDING;
00456     descriptor_object_descriptions_writer.dstArrayElement = 0;
00457     descriptor_object_descriptions_writer.descriptorCount = 1;
00458     descriptor_object_descriptions_writer.descriptorType =
00459         VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00460     descriptor_object_descriptions_writer.pImageInfo = nullptr;
00461     descriptor_object_descriptions_writer.pBufferInfo =
00462         &object_descriptions_buffer_info;
00463     descriptor_object_descriptions_writer.pTexelBufferView =
00464         nullptr;  // information about buffer data to bind
00465
00466     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00467         descriptor_object_descriptions_writer};
00468
00469     // update the descriptor sets with new buffer/binding info
00470     vkUpdateDescriptorSets(device->getLogicalDevice(),
00471                            static_cast<uint32_t>(write_descriptor_sets.size()),
00472                            write_descriptor_sets.data(), 0, nullptr);
00473   }
00474 }
00475
00476 void VulkanRenderer::createRaytracingDescriptorSetLayouts() {
00477   {
00478     std::array<VkDescriptorSetLayoutBinding, 2>
00479         descriptor_set_layout_bindings{};
00480
00481     // here comes the top level acceleration structure
00482     descriptor_set_layout_bindings[0].binding = TLAS_BINDING;
00483     descriptor_set_layout_bindings[0].descriptorCount = 1;
00484     descriptor_set_layout_bindings[0].descriptorType =
00485         VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00486     descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;
00487     // load them into the raygeneration and chlosest hit shader
00488     descriptor_set_layout_bindings[0].stageFlags =
00489         VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00490         VK_SHADER_STAGE_COMPUTE_BIT;
00491     // here comes to previous rendered image
00492     descriptor_set_layout_bindings[1].binding = OUT_IMAGE_BINDING;
00493     descriptor_set_layout_bindings[1].descriptorCount = 1;
00494     descriptor_set_layout_bindings[1].descriptorType =
00495         VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00496     descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;
00497     // load them into the raygeneration and chlosest hit shader
00498     descriptor_set_layout_bindings[1].stageFlags =
00499         VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00500         VK_SHADER_STAGE_COMPUTE_BIT;
00501
00502     VkDescriptorSetLayoutCreateInfo descriptor_set_layout_create_info{};
00503     descriptor_set_layout_create_info.sType =
00504         VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00505     descriptor_set_layout_create_info.bindingCount =
00506         static_cast<uint32_t>(descriptor_set_layout_bindings.size());
00507     descriptor_set_layout_create_info.pBindings =
00508         descriptor_set_layout_bindings.data();
00509
00510     VkResult result = vkCreateDescriptorSetLayout(
00511         device->getLogicalDevice(), &descriptor_set_layout_create_info, nullptr,
00512         &raytracingDescriptorSetLayout);
00513     ASSERT_VULKAN(result, "Failed to create raytracing descriptor set layout!")
00514   }
00515 }
00516
00517 void VulkanRenderer::createRaytracingDescriptorSets() {
```

```
00518   // resize descriptor set list so one for every buffer
00519   raytracingDescriptorSet.resize(vulkanSwapChain.getNumberSwapChainImages());
00520
00521   std::vector<VkDescriptorSetLayout> set_layouts(
00522       vulkanSwapChain.getNumberSwapChainImages(),
00523       raytracingDescriptorSetLayout);
00524
00525   VkDescriptorSetAllocateInfo descriptor_set_allocate_info{};
00526   descriptor_set_allocate_info.sType =
00527       VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00528   ;
00529   descriptor_set_allocate_info.descriptorPool = raytracingDescriptorPool;
00530   descriptor_set_allocate_info.descriptorSetCount =
00531       vulkanSwapChain.getNumberSwapChainImages();
00532   descriptor_set_allocate_info.pSetLayouts = set_layouts.data();
00533
00534   VkResult result = vkAllocateDescriptorSets(device->getLogicalDevice(),
00535                                              &descriptor_set_allocate_info,
00536                                              raytracingDescriptorSet.data());
00537   ASSERT_VULKAN(result, "Failed to allocate raytracing descriptor set!")
00538 }
00539
00540 void VulkanRenderer::updateRaytracingDescriptorSets() {
00541   for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00542     VkWriteDescriptorSetAccelerationStructureKHR
00543         descriptor_set_acceleration_structure{};
00544     descriptor_set_acceleration_structure.sType =
00545         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
00546     descriptor_set_acceleration_structure.pNext = nullptr;
00547     descriptor_set_acceleration_structure.accelerationStructureCount = 1;
00548     VkAccelerationStructureKHR& vulkanTLAS = asManager.getTLAS();
00549     descriptor_set_acceleration_structure.pAccelerationStructures = &vulkanTLAS;
00550
00551     VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
00552     write_descriptor_set_acceleration_structure.sType =
00553         VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00554     write_descriptor_set_acceleration_structure.pNext =
00555         &descriptor_set_acceleration_structure;
00556     write_descriptor_set_acceleration_structure.dstSet =
00557         raytracingDescriptorSet[i];
00558     write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
00559     write_descriptor_set_acceleration_structure.dstArrayElement = 0;
00560     write_descriptor_set_acceleration_structure.descriptorCount = 1;
00561     write_descriptor_set_acceleration_structure.descriptorType =
00562         VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
00563     write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
00564     write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
00565     write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;
00566
00567     VkDescriptorImageInfo image_info{};
00568     Texture& renderResult = rasterizer.getOffscreenTexture(i);
00569     image_info.imageView = renderResult.getImageView();
00570     image_info.imageLayout = VK_IMAGE_LAYOUT_GENERAL;
00571
00572     VkWriteDescriptorSet descriptor_image_writer{};
00573     descriptor_image_writer.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00574     descriptor_image_writer.pNext = nullptr;
00575     descriptor_image_writer.dstSet = raytracingDescriptorSet[i];
00576     descriptor_image_writer.dstBinding = OUT_IMAGE_BINDING;
00577     descriptor_image_writer.dstArrayElement = 0;
00578     descriptor_image_writer.descriptorCount = 1;
00579     descriptor_image_writer.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
00580     descriptor_image_writer.pImageInfo = &image_info;
00581     descriptor_image_writer.pBufferInfo = nullptr;
00582     descriptor_image_writer.pTexelBufferView = nullptr;
00583
00584     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00585         write_descriptor_set_acceleration_structure, descriptor_image_writer};
00586
00587     // update the descriptor sets with new buffer/binding info
00588     vkUpdateDescriptorSets(device->getLogicalDevice(),
00589                            static_cast<uint32_t>(write_descriptor_sets.size()),
00590                            write_descriptor_sets.data(), 0, nullptr);
00591   }
00592 }
00593
00594 void VulkanRenderer::createSharedRenderDescriptorSetLayouts() {
00595   std::array<VkDescriptorSetLayoutBinding, 5> descriptor_set_layout_bindings{};
00596   // UNIFORM VALUES DESCRIPTOR SET LAYOUT
00597   // globalUBO Binding info
00598   descriptor_set_layout_bindings[0].binding = globalUBO_BINDING;
00599   descriptor_set_layout_bindings[0].descriptorType =
00600       VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00601   descriptor_set_layout_bindings[0].descriptorCount = 1;
00602   descriptor_set_layout_bindings[0].stageFlags =
00603       VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_RAYGEN_BIT_KHR |
00604       VK_SHADER_STAGE_COMPUTE_BIT;
```

```
00605   descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;
00606
00607   // our model matrix which updates every frame for each object
00608   descriptor_set_layout_bindings[1].binding = sceneUBO_BINDING;
00609   descriptor_set_layout_bindings[1].descriptorType =
00610       VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00611   descriptor_set_layout_bindings[1].descriptorCount = 1;
00612   descriptor_set_layout_bindings[1].stageFlags =
00613       VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
00614       VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00615       VK_SHADER_STAGE_COMPUTE_BIT;
00616   descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;
00617
00618   descriptor_set_layout_bindings[2].binding = OBJECT_DESCRIPTION_BINDING;
00619   descriptor_set_layout_bindings[2].descriptorCount = 1;
00620   descriptor_set_layout_bindings[2].descriptorType =
00621       VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00622   descriptor_set_layout_bindings[2].pImmutableSamplers = nullptr;
00623   // load them into the raygeneration and chlosest hit shader
00624   descriptor_set_layout_bindings[2].stageFlags =
00625       VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
00626       VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR | VK_SHADER_STAGE_COMPUTE_BIT;
00627
00628   // CREATE TEXTURE SAMPLER DESCRIPTOR SET LAYOUT
00629   // texture binding info
00630   descriptor_set_layout_bindings[3].binding = SAMPLER_BINDING;
00631   descriptor_set_layout_bindings[3].descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
00632   descriptor_set_layout_bindings[3].descriptorCount = MAX_TEXTURE_COUNT;
00633   descriptor_set_layout_bindings[3].stageFlags =
00634       VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00635       VK_SHADER_STAGE_COMPUTE_BIT;
00636   descriptor_set_layout_bindings[3].pImmutableSamplers = nullptr;
00637
00638   descriptor_set_layout_bindings[4].binding = TEXTURES_BINDING;
00639   descriptor_set_layout_bindings[4].descriptorType =
00640       VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00641   descriptor_set_layout_bindings[4].descriptorCount = MAX_TEXTURE_COUNT;
00642   descriptor_set_layout_bindings[4].stageFlags =
00643       VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
00644       VK_SHADER_STAGE_COMPUTE_BIT;
00645   descriptor_set_layout_bindings[4].pImmutableSamplers = nullptr;
00646
00647   // create descriptor set layout with given bindings
00648   VkDescriptorSetLayoutCreateInfo layout_create_info{};
00649   layout_create_info.sType =
00650       VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
00651   layout_create_info.bindingCount =
00652       static_cast<uint32_t>(descriptor_set_layout_bindings.size());
00653   layout_create_info.pBindings = descriptor_set_layout_bindings.data();
00654
00655   // create descriptor set layout
00656   VkResult result = vkCreateDescriptorSetLayout(
00657       device->getLogicalDevice(), &layout_create_info, nullptr,
00658       &sharedRenderDescriptorSetLayout);
00659   ASSERT_VULKAN(result, "Failed to create descriptor set layout!")
00660 }
00661
00662 void VulkanRenderer::create_command_pool() {
00663   // get indices of queue familes from device
00664   QueueFamilyIndices queue_family_indices = device->getQueueFamilies();
00665
00666   {
00667     VkCommandPoolCreateInfo pool_info{};
00668     pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
00669     pool_info.flags =
00670         VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;  // we are ready now to
00671                                                           // re-record our
00672                                                           // command buffers
00673     pool_info.queueFamilyIndex =
00674         queue_family_indices
00675             .graphics_family;  // queue family type that buffers from this
00676                                // command pool will use
00677
00678     // create a graphics queue family command pool
00679     VkResult result =
00680         vkCreateCommandPool(device->getLogicalDevice(), &pool_info, nullptr,
00681                             &graphics_command_pool);
00682     ASSERT_VULKAN(result, "Failed to create command pool!")
00683   }
00684
00685   {
00686     VkCommandPoolCreateInfo pool_info{};
00687     pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
00688     pool_info.flags =
00689         VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;  // we are ready now to
00690                                                           // re-record our
00691                                                           // command buffers
```

```
00692     pool_info.queueFamilyIndex =
00693         queue_family_indices.compute_family;  // queue family type that buffers
00694                                               // from this command pool will use
00695
00696     // create a graphics queue family command pool
00697     VkResult result = vkCreateCommandPool(
00698         device->getLogicalDevice(), &pool_info, nullptr, &compute_command_pool);
00699     ASSERT_VULKAN(result, "Failed to create command pool!")
00700   }
00701 }
00702
00703 void VulkanRenderer::cleanUpCommandPools() {
00704   vkDestroyCommandPool(device->getLogicalDevice(), graphics_command_pool,
00705                        nullptr);
00706   vkDestroyCommandPool(device->getLogicalDevice(), compute_command_pool,
00707                        nullptr);
00708 }
00709
00710 void VulkanRenderer::create_command_buffers() {
00711   // resize command buffer count to have one for each framebuffer
00712   command_buffers.resize(vulkanSwapChain.getNumberSwapChainImages());
00713
00714   VkCommandBufferAllocateInfo command_buffer_alloc_info{};
00715   command_buffer_alloc_info.sType =
00716       VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
00717   command_buffer_alloc_info.commandPool = graphics_command_pool;
00718   command_buffer_alloc_info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
00719
00720   command_buffer_alloc_info.commandBufferCount =
00721       static_cast<uint32_t>(command_buffers.size());
00722
00723   VkResult result = vkAllocateCommandBuffers(device->getLogicalDevice(),
00724                                              &command_buffer_alloc_info,
00725                                              command_buffers.data());
00726   ASSERT_VULKAN(result, "Failed to allocate command buffers!")
00727 }
00728
00729 void VulkanRenderer::createSynchronization() {
00730   image_available.resize(vulkanSwapChain.getNumberSwapChainImages(),
00731                          VK_NULL_HANDLE);
00732   render_finished.resize(vulkanSwapChain.getNumberSwapChainImages(),
00733                          VK_NULL_HANDLE);
00734   in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
00735                           VK_NULL_HANDLE);
00736   images_in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
00737                                  VK_NULL_HANDLE);
00738
00739   // semaphore creation information
00740   VkSemaphoreCreateInfo semaphore_create_info{};
00741   semaphore_create_info.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
00742
00743   // fence creation information
00744   VkFenceCreateInfo fence_create_info{};
00745   fence_create_info.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
00746   fence_create_info.flags = VK_FENCE_CREATE_SIGNALED_BIT;
00747
00748   for (int i = 0; i < MAX_FRAME_DRAWS; i++) {
00749     if ((vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
00750                            nullptr, &image_available[i]) != VK_SUCCESS) ||
00751         (vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
00752                            nullptr, &render_finished[i]) != VK_SUCCESS) ||
00753         (vkCreateFence(device->getLogicalDevice(), &fence_create_info, nullptr,
00754                        &in_flight_fences[i]) != VK_SUCCESS)) {
00755       throw std::runtime_error("Failed to create a semaphore and/or fence!");
00756     }
00757   }
00758 }
00759
00760 void VulkanRenderer::create_uniform_buffers() {
00761   // one uniform buffer for each image (and by extension, command buffer)
00762   globalUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());
00763   sceneUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());
00764
00765   //// temporary buffer to "stage" vertex data before transfering to GPU
00766   // VulkanBuffer       stagingBuffer;
00767   std::vector<GlobalUBO> globalUBOdata;
00768   globalUBOdata.push_back(globalUBO);
00769
00770   std::vector<SceneUBO> sceneUBOdata;
00771   sceneUBOdata.push_back(sceneUBO);
00772
00773   // create uniform buffers
00774   for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00775     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00776         device.get(), graphics_command_pool, globalUBOBuffer[i],
00777         VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00778         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, globalUBOdata);
```

```
00779
00780     vulkanBufferManager.createBufferAndUploadVectorOnDevice(
00781         device.get(), graphics_command_pool, sceneUBOBuffer[i],
00782         VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
00783         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, sceneUBOdata);
00784   }
00785 }
00786
00787 void VulkanRenderer::createDescriptorPoolSharedRenderStages() {
00788   // CREATE UNIFORM DESCRIPTOR POOL
00789   // type of descriptors + how many descriptors, not descriptor sets (combined
00790   // makes the pool size) ViewProjection Pool
00791   VkDescriptorPoolSize vp_pool_size{};
00792   vp_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00793   vp_pool_size.descriptorCount = static_cast<uint32_t>(globalUBOBuffer.size());
00794
00795   // DIRECTION POOL
00796   VkDescriptorPoolSize directions_pool_size{};
00797   directions_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
00798   directions_pool_size.descriptorCount =
00799       static_cast<uint32_t>(sceneUBOBuffer.size());
00800
00801   VkDescriptorPoolSize object_descriptions_pool_size{};
00802   object_descriptions_pool_size.type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
00803   object_descriptions_pool_size.descriptorCount =
00804       static_cast<uint32_t>(sizeof(ObjectDescription) * MAX_OBJECTS);
00805
00806   // TEXTURE SAMPLER POOL
00807   VkDescriptorPoolSize sampler_pool_size{};
00808   sampler_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLER;
00809   sampler_pool_size.descriptorCount = MAX_TEXTURE_COUNT;
00810
00811   VkDescriptorPoolSize sampled_image_pool_size{};
00812   sampled_image_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00813   sampled_image_pool_size.descriptorCount = MAX_TEXTURE_COUNT;
00814
00815   // list of pool sizes
00816   std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {
00817       vp_pool_size, directions_pool_size, object_descriptions_pool_size,
00818       sampler_pool_size, sampled_image_pool_size};
00819
00820   VkDescriptorPoolCreateInfo pool_create_info{};
00821   pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
00822   pool_create_info.maxSets =
00823       vulkanSwapChain
00824           .getNumberSwapChainImages();  // maximum number of descriptor sets
00825                                         // that can be created from pool
00826   pool_create_info.poolSizeCount = static_cast<uint32_t>(
00827       descriptor_pool_sizes.size());  // amount of pool sizes being passed
00828   pool_create_info.pPoolSizes =
00829       descriptor_pool_sizes.data();  // pool sizes to create pool with
00830
00831   // create descriptor pool
00832   VkResult result =
00833       vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
00834                              nullptr, &descriptorPoolSharedRenderStages);
00835   ASSERT_VULKAN(result, "Failed to create a descriptor pool!")
00836 }
00837
00838 void VulkanRenderer::createSharedRenderDescriptorSet() {
00839   // resize descriptor set list so one for every buffer
00840   sharedRenderDescriptorSet.resize(vulkanSwapChain.getNumberSwapChainImages());
00841
00842   std::vector<VkDescriptorSetLayout> set_layouts(
00843       vulkanSwapChain.getNumberSwapChainImages(),
00844       sharedRenderDescriptorSetLayout);
00845
00846   // descriptor set allocation info
00847   VkDescriptorSetAllocateInfo set_alloc_info{};
00848   set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
00849   set_alloc_info.descriptorPool =
00850       descriptorPoolSharedRenderStages;  // pool to allocate descriptor set from
00851   set_alloc_info.descriptorSetCount =
00852       vulkanSwapChain.getNumberSwapChainImages();  // number of sets to allocate
00853   set_alloc_info.pSetLayouts =
00854       set_layouts.data();  // layouts to use to allocate sets (1:1 relationship)
00855
00856   // allocate descriptor sets (multiple)
00857   VkResult result =
00858       vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
00859                                sharedRenderDescriptorSet.data());
00860   ASSERT_VULKAN(result, "Failed to create descriptor sets!")
00861
00862   // update all of descriptor set buffer bindings
00863   for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00864     // VIEW PROJECTION DESCRIPTOR
00865     // buffer info and data offset info
```

```
00866     VkDescriptorBufferInfo globalUBO_buffer_info{};
00867     globalUBO_buffer_info.buffer =
00868         globalUBOBuffer[i].getBuffer();  // buffer to get data from
00869     globalUBO_buffer_info.offset = 0;    // position of start of data
00870     globalUBO_buffer_info.range = sizeof(globalUBO);  // size of data
00871
00872     // data about connection between binding and buffer
00873     VkWriteDescriptorSet globalUBO_set_write{};
00874     globalUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00875     globalUBO_set_write.dstSet =
00876         sharedRenderDescriptorSet[i];  // descriptor set to update
00877     globalUBO_set_write.dstBinding =
00878         0;  // binding to update (matches with binding on layout/shader)
00879     globalUBO_set_write.dstArrayElement = 0;  // index in array to update
00880     globalUBO_set_write.descriptorType =
00881         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;    // type of descriptor
00882     globalUBO_set_write.descriptorCount = 1;  // amount to update
00883     globalUBO_set_write.pBufferInfo =
00884         &globalUBO_buffer_info;  // information about buffer data to bind
00885
00886     // VIEW PROJECTION DESCRIPTOR
00887     // buffer info and data offset info
00888     VkDescriptorBufferInfo sceneUBO_buffer_info{};
00889     sceneUBO_buffer_info.buffer =
00890         sceneUBOBuffer[i].getBuffer();              // buffer to get data from
00891     sceneUBO_buffer_info.offset = 0;                // position of start of data
00892     sceneUBO_buffer_info.range = sizeof(sceneUBO);  // size of data
00893
00894     // data about connection between binding and buffer
00895     VkWriteDescriptorSet sceneUBO_set_write{};
00896     sceneUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00897     sceneUBO_set_write.dstSet =
00898         sharedRenderDescriptorSet[i];  // descriptor set to update
00899     sceneUBO_set_write.dstBinding =
00900         1;  // binding to update (matches with binding on layout/shader)
00901     sceneUBO_set_write.dstArrayElement = 0;  // index in array to update
00902     sceneUBO_set_write.descriptorType =
00903         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;   // type of descriptor
00904     sceneUBO_set_write.descriptorCount = 1;  // amount to update
00905     sceneUBO_set_write.pBufferInfo =
00906         &sceneUBO_buffer_info;  // information about buffer data to bind
00907
00908     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00909         globalUBO_set_write, sceneUBO_set_write};
00910
00911     // update the descriptor sets with new buffer/binding info
00912     vkUpdateDescriptorSets(device->getLogicalDevice(),
00913                            static_cast<uint32_t>(write_descriptor_sets.size()),
00914                            write_descriptor_sets.data(), 0, nullptr);
00915   }
00916 }
00917
00918 void VulkanRenderer::updateTexturesInSharedRenderDescriptorSet() {
00919   std::vector<Texture>& modelTextures = scene->getTextures(0);
00920   std::vector<VkDescriptorImageInfo> image_info_textures;
00921   image_info_textures.resize(scene->getTextureCount(0));
00922   for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
00923     image_info_textures[i].imageLayout =
00924         VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00925     image_info_textures[i].imageView = modelTextures[i].getImageView();
00926     image_info_textures[i].sampler = nullptr;
00927   }
00928
00929   std::vector<VkSampler>& modelTextureSampler = scene->getTextureSampler(0);
00930   std::vector<VkDescriptorImageInfo> image_info_texture_sampler;
00931   image_info_texture_sampler.resize(scene->getTextureCount(0));
00932   for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
00933     image_info_texture_sampler[i].imageLayout =
00934         VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
00935     image_info_texture_sampler[i].imageView = nullptr;
00936     image_info_texture_sampler[i].sampler = modelTextureSampler[i];
00937   }
00938
00939   for (uint32_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
00940     // descriptor write info
00941     VkWriteDescriptorSet descriptor_write{};
00942     descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00943     descriptor_write.dstSet = sharedRenderDescriptorSet[i];
00944     descriptor_write.dstBinding = TEXTURES_BINDING;
00945     descriptor_write.dstArrayElement = 0;
00946     descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
00947     descriptor_write.descriptorCount =
00948         static_cast<uint32_t>(image_info_textures.size());
00949     descriptor_write.pImageInfo = image_info_textures.data();
00950
00951     /*VkDescriptorImageInfo sampler_info;
00952                 sampler_info.imageView = nullptr;
```

```
00953                 sampler_info.sampler = texture_sampler;*/
00954
00955     // descriptor write info
00956     VkWriteDescriptorSet descriptor_write_sampler{};
00957     descriptor_write_sampler.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
00958     descriptor_write_sampler.dstSet = sharedRenderDescriptorSet[i];
00959     descriptor_write_sampler.dstBinding = SAMPLER_BINDING;
00960     descriptor_write_sampler.dstArrayElement = 0;
00961     descriptor_write_sampler.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
00962     descriptor_write_sampler.descriptorCount =
00963         static_cast<uint32_t>(image_info_texture_sampler.size());
00964     descriptor_write_sampler.pImageInfo = image_info_texture_sampler.data();
00965
00966     std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
00967         descriptor_write, descriptor_write_sampler};
00968
00969     // update new descriptor set
00970     vkUpdateDescriptorSets(device->getLogicalDevice(),
00971                            static_cast<uint32_t>(write_descriptor_sets.size()),
00972                            write_descriptor_sets.data(), 0, nullptr);
00973   }
00974 }
00975
00976 void VulkanRenderer::cleanUpUBOs() {
00977   for (VulkanBuffer vulkanBuffer : globalUBOBuffer) {
00978     vulkanBuffer.cleanUp();
00979   }
00980
00981   for (VulkanBuffer vulkanBuffer : sceneUBOBuffer) {
00982     vulkanBuffer.cleanUp();
00983   }
00984 }
00985
00986 void VulkanRenderer::update_uniform_buffers(uint32_t image_index) {
00987   auto usage_stage_flags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
00988                            VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR |
00989                            VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
00990
00991   VkBufferMemoryBarrier before_barrier_uvp{};
00992   before_barrier_uvp.pNext = nullptr;
00993   before_barrier_uvp.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
00994   before_barrier_uvp.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
00995   before_barrier_uvp.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
00996   before_barrier_uvp.buffer = globalUBOBuffer[image_index].getBuffer();
00997   before_barrier_uvp.offset = 0;
00998   before_barrier_uvp.size = sizeof(globalUBO);
00999   before_barrier_uvp.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01000   before_barrier_uvp.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01001
01002   VkBufferMemoryBarrier before_barrier_directions{};
01003   before_barrier_directions.pNext = nullptr;
01004   before_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01005   before_barrier_directions.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
01006   before_barrier_directions.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01007   before_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
01008   before_barrier_directions.offset = 0;
01009   before_barrier_directions.size = sizeof(sceneUBO);
01010   before_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01011   before_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01012
01013   vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
01014                        VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
01015                        &before_barrier_uvp, 0, nullptr);
01016   vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
01017                        VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
01018                        &before_barrier_directions, 0, nullptr);
01019
01020   vkCmdUpdateBuffer(command_buffers[image_index],
01021                     globalUBOBuffer[image_index].getBuffer(), 0,
01022                     sizeof(GlobalUBO), &globalUBO);
01023   vkCmdUpdateBuffer(command_buffers[image_index],
01024                     sceneUBOBuffer[image_index].getBuffer(), 0,
01025                     sizeof(SceneUBO), &sceneUBO);
01026
01027   VkBufferMemoryBarrier after_barrier_uvp{};
01028   after_barrier_uvp.pNext = nullptr;
01029   after_barrier_uvp.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01030   after_barrier_uvp.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01031   after_barrier_uvp.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
01032   after_barrier_uvp.buffer = globalUBOBuffer[image_index].getBuffer();
01033   after_barrier_uvp.offset = 0;
01034   after_barrier_uvp.size = sizeof(GlobalUBO);
01035   after_barrier_uvp.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01036   after_barrier_uvp.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01037
01038   VkBufferMemoryBarrier after_barrier_directions{};
01039   after_barrier_directions.pNext = nullptr;
```

```
01040   after_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
01041   after_barrier_directions.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
01042   after_barrier_directions.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
01043   after_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
01044   after_barrier_directions.offset = 0;
01045   after_barrier_directions.size = sizeof(SceneUBO);
01046   after_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01047   after_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
01048
01049   vkCmdPipelineBarrier(command_buffers[image_index],
01050                        VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
01051                        nullptr, 1, &after_barrier_uvp, 0, nullptr);
01052   vkCmdPipelineBarrier(command_buffers[image_index],
01053                        VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
01054                        nullptr, 1, &after_barrier_directions, 0, nullptr);
01055 }
01056
01057 void VulkanRenderer::update_raytracing_descriptor_set(uint32_t image_index) {
01058   VkWriteDescriptorSetAccelerationStructureKHR
01059       descriptor_set_acceleration_structure{};
01060   descriptor_set_acceleration_structure.sType =
01061       VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
01062   descriptor_set_acceleration_structure.pNext = nullptr;
01063   descriptor_set_acceleration_structure.accelerationStructureCount = 1;
01064   VkAccelerationStructureKHR& tlasAS = asManager.getTLAS();
01065   descriptor_set_acceleration_structure.pAccelerationStructures = &tlasAS;
01066
01067   VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
01068   write_descriptor_set_acceleration_structure.sType =
01069       VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
01070   write_descriptor_set_acceleration_structure.pNext =
01071       &descriptor_set_acceleration_structure;
01072   write_descriptor_set_acceleration_structure.dstSet =
01073       raytracingDescriptorSet[image_index];
01074   write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
01075   write_descriptor_set_acceleration_structure.dstArrayElement = 0;
01076   write_descriptor_set_acceleration_structure.descriptorCount = 1;
01077   write_descriptor_set_acceleration_structure.descriptorType =
01078       VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
01079   write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
01080   write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
01081   write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;
01082
01083   VkDescriptorBufferInfo object_description_buffer_info{};
01084   object_description_buffer_info.buffer = objectDescriptionBuffer.getBuffer();
01085   object_description_buffer_info.offset = 0;
01086   object_description_buffer_info.range = VK_WHOLE_SIZE;
01087
01088   VkWriteDescriptorSet object_description_buffer_write{};
01089   object_description_buffer_write.sType =
01090       VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
01091   object_description_buffer_write.dstSet =
01092       sharedRenderDescriptorSet[image_index];
01093   object_description_buffer_write.descriptorType =
01094       VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
01095   object_description_buffer_write.dstBinding = OBJECT_DESCRIPTION_BINDING;
01096   object_description_buffer_write.pBufferInfo = &object_description_buffer_info;
01097   object_description_buffer_write.descriptorCount = 1;
01098
01099   std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
01100       write_descriptor_set_acceleration_structure,
01101       object_description_buffer_write};
01102
01103   vkUpdateDescriptorSets(device->getLogicalDevice(),
01104                          static_cast<uint32_t>(write_descriptor_sets.size()),
01105                          write_descriptor_sets.data(), 0, nullptr);
01106 }
01107
01108 void VulkanRenderer::record_commands(uint32_t image_index) {
01109   Texture& renderResult = rasterizer.getOffscreenTexture(image_index);
01110   VulkanImage& vulkanImage = renderResult.getVulkanImage();
01111
01112   GUIRendererSharedVars& guiRendererSharedVars =
01113       gui->getGuiRendererSharedVars();
01114   if (guiRendererSharedVars.raytracing) {
01115     std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],
01116                                          raytracingDescriptorSet[image_index]};
01117     raytracingStage.recordCommands(command_buffers[image_index],
01118                                    &vulkanSwapChain, sets);
01119
01120   } else if (guiRendererSharedVars.pathTracing) {
01121     std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],
01122                                          raytracingDescriptorSet[image_index]};
01123
01124     pathTracing.recordCommands(command_buffers[image_index], image_index,
01125                                vulkanImage, &vulkanSwapChain, sets);
01126
```

```
01127   } else {
01128     std::vector<VkDescriptorSet> descriptorSets = {
01129         sharedRenderDescriptorSet[image_index]};
01130
01131     rasterizer.recordCommands(command_buffers[image_index], image_index, scene,
01132                               descriptorSets);
01133   }
01134
01135   vulkanImage.transitionImageLayout(
01136       command_buffers[image_index], VK_IMAGE_LAYOUT_GENERAL,
01137       VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);
01138
01139   std::vector<VkDescriptorSet> descriptorSets = {
01140       post_descriptor_set[image_index]};
01141   postStage.recordCommands(command_buffers[image_index], image_index,
01142                            descriptorSets);
01143
01144   vulkanImage.transitionImageLayout(
01145       command_buffers[image_index], VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
01146       VK_IMAGE_LAYOUT_GENERAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);
01147 }
01148
01149 bool VulkanRenderer::checkChangedFramebufferSize() {
01150   if (window->framebuffer_size_has_changed()) {
01151     vkDeviceWaitIdle(device->getLogicalDevice());
01152     vkQueueWaitIdle(device->getGraphicsQueue());
01153
01154     vulkanSwapChain.cleanUp();
01155     vulkanSwapChain.initVulkanContext(device.get(), window, surface);
01156
01157     std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
01158         sharedRenderDescriptorSetLayout};
01159     rasterizer.cleanUp();
01160     rasterizer.init(device.get(), &vulkanSwapChain, descriptor_set_layouts,
01161                     graphics_command_pool);
01162
01163     // all post
01164     std::vector<VkDescriptorSetLayout> descriptorSets = {
01165         post_descriptor_set_layout};
01166     postStage.cleanUp();
01167     postStage.init(device.get(), &vulkanSwapChain, descriptorSets);
01168
01169     gui->cleanUp();
01170     gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
01171                                  postStage.getRenderPass(),
01172                                  graphics_command_pool);
01173
01174     current_frame = 0;
01175
01176     updatePostDescriptorSets();
01177     updateRaytracingDescriptorSets();
01178
01179     window->reset_framebuffer_has_changed();
01180
01181     return true;
01182   }
01183
01184   return false;
01185 }
01186
01187 void VulkanRenderer::cleanUp() {
01188   cleanUpUBOs();
01189
01190   rasterizer.cleanUp();
01191   raytracingStage.cleanUp();
01192   postStage.cleanUp();
01193   pathTracing.cleanUp();
01194
01195   objectDescriptionBuffer.cleanUp();
01196   asManager.cleanUp();
01197
01198   vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01199                                raytracingDescriptorSetLayout, nullptr);
01200   vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01201                                post_descriptor_set_layout, nullptr);
01202   vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
01203                                sharedRenderDescriptorSetLayout, nullptr);
01204   vkDestroyDescriptorPool(device->getLogicalDevice(), post_descriptor_pool,
01205                           nullptr);
01206   vkDestroyDescriptorPool(device->getLogicalDevice(),
01207                           descriptorPoolSharedRenderStages, nullptr);
01208   vkDestroyDescriptorPool(device->getLogicalDevice(), raytracingDescriptorPool,
01209                           nullptr);
01210
01211   vkFreeCommandBuffers(device->getLogicalDevice(), graphics_command_pool,
01212                        static_cast<uint32_t>(command_buffers.size()),
01213                        command_buffers.data());
```

```
01214
01215   cleanUpCommandPools();
01216
01217   cleanUpSync();
01218
01219   vulkanSwapChain.cleanUp();
01220   vkDestroySurfaceKHR(instance.getVulkanInstance(), surface, nullptr);
01221   allocator.cleanUp();
01222   device->cleanUp();
01223   debug::freeDebugCallback(instance.getVulkanInstance());
01224   instance.cleanUp();
01225 }
01226
01227 VulkanRenderer::~VulkanRenderer() {}
```

4.23 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/Camera.cpp File Reference

`#include "Camera.h"`
Include dependency graph for Camera.cpp:

4.24 Camera.cpp

Go to the documentation of this file.

```
00001 #include "Camera.h"
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.f)),
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.f),
00012       pitch(-40.0f),
00013       movement_speed(200.f),
00014       turn_speed(0.25f),
00015       near_plane(0.1f),
00016       far_plane(4000.f),
00017       fov(45.f)
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_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; pich 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 }
```

4.25 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/↩ Mesh.cpp File Reference

```
#include "Mesh.h"
#include <cstring>
#include <memory>
#include "VulkanBuffer.h"
```

Include dependency graph for Mesh.cpp:

4.26 Mesh.cpp

Go to the documentation of this file.

```
00001 #include "Mesh.h"
```

```
00002
00003 #include <cstring>
00004 #include <memory>
00005
00006 #include "VulkanBuffer.h"
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 }
```

4.27 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/↩ Model.cpp File Reference

`#include "Model.h"`

Include dependency graph for Model.cpp:

4.28 Model.cpp

Go to the documentation of this file.

```
00001 #include "Model.h"
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
00041         number_of_indices += mesh.get_index_count();
00042
00043     }
00044
00045     return number_of_indices / 3;*/
00046   return mesh.getIndexCount() / 3;
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 }
```

4.29 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/ObjLoader.cpp File Reference

```
#include "ObjLoader.h"
#include <tiny_obj_loader.h>
#include "File.h"
```

Include dependency graph for ObjLoader.cpp:

Macros

- #define TINYOBJLOADER_IMPLEMENTATION

4.29.1 Macro Definition Documentation

4.29.1.1 TINYOBJLOADER_IMPLEMENTATION

```
#define TINYOBJLOADER_IMPLEMENTATION
```

Definition at line 2 of file ObjLoader.cpp.

4.30 ObjLoader.cpp

Go to the documentation of this file.

```
00001 #include "ObjLoader.h"
00002 #define TINYOBJLOADER_IMPLEMENTATION
00003 #include <tiny_obj_loader.h>
00004
00005 #include "File.h"
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 txtures 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         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 }
```

4.31 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/↩ Scene.cpp File Reference

`#include "Scene.h"`

Include dependency graph for Scene.cpp:

4.32 Scene.cpp

Go to the documentation of this file.

```
00001 #include "Scene.h"
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
00015   add_model(new_model);
00016
00017   glm::mat4 modelMatrix = sceneConfig::getModelMatrix();
00018
00019   update_model_matrix(modelMatrix, 0);
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() {}
```

4.33 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/SceneConfig.cpp File Reference

`#include <SceneConfig.h>`
Include dependency graph for SceneConfig.cpp:

Namespaces

- namespace sceneConfig

Functions

- std::string sceneConfig::getModelFile ()
- glm::mat4 sceneConfig::getModelMatrix ()

4.34 SceneConfig.cpp

Go to the documentation of this file.

```
00001 #include <SceneConfig.h>
00002
00003 //#define SULO_MODE 0
00004
00005 namespace sceneConfig {
00006
00007 std::string getModelFile() {
00008   std::stringstream modelFile;
00009   modelFile « CMAKELISTS_DIR;
00010 #if NDEBUG
00011   modelFile « "/Resources/Model/crytek-sponza/";
00012   modelFile « "sponza_triag.obj";
00013
00014 #else
00015 #ifdef SULO_MODE
00016   modelFile « "/Resources/Model/Sulo/";
00017   modelFile « "SuloLongDongLampe_v2.obj";
00018 #else
00019   modelFile « "/Resources/Model/VikingRoom/";
00020   modelFile « "viking_room.obj";
00021 #endif
00022 #endif
00023
00024   return modelFile.str();
00025   // std::string modelFile =
00026   // "../Resources/Model/crytek-sponza/sponza_triag.obj"; std::string modelFile
00027   // = "../Resources/Model/Dinosaurs/dinosaurs.obj"; std::string modelFile =
00028   // "../Resources/Model/Pillum/PilumPainting_export.obj"; std::string modelFile
00029   // = "../Resources/Model/sibenik/sibenik.obj"; std::string modelFile =
00030   // "../Resources/Model/sportsCar/sportsCar.obj"; std::string modelFile =
00031   // "../Resources/Model/StanfordDragon/dragon.obj"; std::string modelFile =
00032   // "../Resources/Model/CornellBox/CornellBox-Sphere.obj"; std::string
00033   // modelFile = "../Resources/Model/bunny/bunny.obj"; std::string modelFile =
00034   // "../Resources/Model/buddha/buddha.obj"; std::string modelFile =
00035   // "../Resources/Model/bmw/bmw.obj"; std::string modelFile =
00036   // "../Resources/Model/testScene.obj"; std::string modelFile =
00037   // "../Resources/Model/San_Miguel/san-miguel-low-poly.obj";
00038 }
00039
00040 glm::mat4 getModelMatrix() {
00041   glm::mat4 modelMatrix(1.0f);
00042
00043 #if NDEBUG
00044
00045   // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00046   // -50.0f));
00047   modelMatrix = glm::scale(modelMatrix, glm::vec3(1.0f, 1.0f, 1.0f));
00048   /*dragon_model = glm::rotate(dragon_model, glm::radians(-90.f),
00049      glm::vec3(1.0f, 0.0f, 0.0f)); dragon_model = glm::rotate(dragon_model,
00050      glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));*/
00051
00052 #else
00053
00054 // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
00055 // -50.0f));
00056 #if SULO_MODE
00057   modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00058 #else
00059   modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
00060   modelMatrix = glm::rotate(modelMatrix, glm::radians(-90.f),
00061                             glm::vec3(1.0f, 0.0f, 0.0f));
00062   modelMatrix =
00063       glm::rotate(modelMatrix, glm::radians(90.f), glm::vec3(0.0f, 0.0f, 1.0f));
00064 #endif
00065
00066 #endif
00067
00068   return modelMatrix;
00069 }
00070
00071 }  // namespace sceneConfig
```

4.35 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/↩ Texture.cpp File Reference

```
#include "Texture.h"
#include <cmath>
```

#include <stdexcept>
Include dependency graph for Texture.cpp:

4.36 Texture.cpp

Go to the documentation of this file.

```
00001 #include "Texture.h"
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) {
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 }
00062
00063 void Texture::setImage(VkImage image) { vulkanImage.setImage(image); }
00064
00065 void Texture::setImageView(VkImageView imageView) {
00066   vulkanImageView.setImageView(imageView);
00067 }
00068
00069 void Texture::createImage(VulkanDevice* device, uint32_t width, uint32_t height,
00070                           uint32_t mip_levels, VkFormat format,
00071                           VkImageTiling tiling, VkImageUsageFlags use_flags,
00072                           VkMemoryPropertyFlags prop_flags) {
00073   vulkanImage.create(device, width, height, mip_levels, format, tiling,
00074                      use_flags, prop_flags);
00075 }
```

```
00076
00077 void Texture::createImageView(VulkanDevice* device, VkFormat format,
00078                               VkImageAspectFlags aspect_flags,
00079                               uint32_t mip_levels) {
00080   vulkanImageView.create(device, vulkanImage.getImage(), format, aspect_flags,
00081                          mip_levels);
00082 }
00083
00084 void Texture::cleanUp() {
00085   vulkanImageView.cleanUp();
00086   vulkanImage.cleanUp();
00087 }
00088
00089 Texture::~Texture() {}
00090
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
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 }
```

4.37 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/scene/Vertex.cpp File Reference

`#include "Vertex.h"`
Include dependency graph for Vertex.cpp:

Namespaces

- namespace vertex

Functions

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

4.38 Vertex.cpp

Go to the documentation of this file.

```
00001 #include "Vertex.h"
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_describtions;
00022
00023   // Position attribute
00024   attribute_describtions[0].binding = 0;
00025   attribute_describtions[0].location = 0;
00026   attribute_describtions[0].format =
00027       VK_FORMAT_R32G32B32_SFLOAT;  // format data will take (also helps define
00028                                    // size of data)
00029   attribute_describtions[0].offset = offsetof(Vertex, pos);
00030
00031   // normal coord attribute
00032   attribute_describtions[1].binding = 0;
00033   attribute_describtions[1].location = 1;
00034   attribute_describtions[1].format =
00035       VK_FORMAT_R32G32B32_SFLOAT;  // format data will take (also helps define
00036                                    // size of data)
00037   attribute_describtions[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_describtions[2].binding = 0;
00043   attribute_describtions[2].location = 2;
00044   attribute_describtions[2].format =
00045       VK_FORMAT_R32G32B32_SFLOAT;  // format data will take (also helps define
00046                                    // size of data)
00047   attribute_describtions[2].offset = offsetof(Vertex, color);
00048
00049   attribute_describtions[3].binding = 0;
00050   // texture coord attribute
00051   attribute_describtions[3].location = 3;
00052   attribute_describtions[3].format =
00053       VK_FORMAT_R32G32_SFLOAT;  // format data will take (also helps define size
00054                                 // of data)
00055   attribute_describtions[3].offset =
00056       offsetof(Vertex, texture_coords);  // where this attribute is defined in
00057                                          // the data for a single vertex
00058
00059   return attribute_describtions;
00060 }
00061
00062 }  // namespace vertex
```

4.39 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/util/File.cpp File Reference

```
#include "File.h"
#include <fstream>
#include <iostream>
```

Include dependency graph for File.cpp:

4.40 File.cpp

Go to the documentation of this file.

```
00001 #include "File.h"
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 %s. 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() {}
```

4.41 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ ShaderHelper.cpp File Reference

```
#include "ShaderHelper.h"
#include <sstream>
#include "Utilities.h"
```

Include dependency graph for ShaderHelper.cpp:

4.42 ShaderHelper.cpp

Go to the documentation of this file.

```
00001 #include "ShaderHelper.h"
00002
00003 #include <sstream>
00004
00005 #include "Utilities.h"
00006
00007 ShaderHelper::ShaderHelper() {}
00008
00009 void ShaderHelper::compileShader(const std::string& shader_src_dir,
00010                                  const std::string& shader_name) {
00011   // GLSLC_EXE is set by cmake to the location of the vulkan glslc
00012   std::stringstream shader_src_path;
00013   std::stringstream shader_log_file;
00014   std::stringstream cmdShaderCompile;
00015   std::stringstream adminPriviliges;
00016   adminPriviliges « "runas /user:<admin-user> \"";
00017
00018   shader_src_path « shader_src_dir « shader_name;
00019   std::string shader_spv_path = getShaderSpvDir(shader_src_dir, shader_name);
00020   shader_log_file « shader_src_dir « shader_name « ".log.txt";
00021   std::stringstream log_stdout_and_stderr;
00022   log_stdout_and_stderr « " > " « shader_log_file.str() « " 2> "
00023                         « shader_log_file.str();
00024
00025   cmdShaderCompile  //« adminPriviliges.str()
00026       « GLSLC_EXE « target « shader_src_path.str() « " -o "
00027       « shader_spv_path;
00028   //« log_stdout_and_stderr.str();
00029
00030   // std::cout « cmdShaderCompile.str().c_str();
00031
00032   system(cmdShaderCompile.str().c_str());
00033 }
00034
00035 std::string ShaderHelper::getShaderSpvDir(const std::string& shader_src_dir,
00036                                           const std::string& shader_name) {
00037   std::string shader_spv_dir = "spv/";
00038
00039   std::stringstream vertShaderSpv;
00040   vertShaderSpv « shader_src_dir « shader_spv_dir « shader_name « ".spv";
00041
00042   return vertShaderSpv.str();
00043 }
00044
00045 VkShaderModule ShaderHelper::createShaderModule(VulkanDevice* device,
00046                                                 const std::vector<char>& code) {
00047   // shader module create info
00048   VkShaderModuleCreateInfo shader_module_create_info{};
00049   shader_module_create_info.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
00050   shader_module_create_info.codeSize = code.size();  // size of code
00051   shader_module_create_info.pCode =
00052       reinterpret_cast<const uint32_t*>(code.data());  // pointer to code
00053
00054   VkShaderModule shader_module;
00055   VkResult result =
00056       vkCreateShaderModule(device->getLogicalDevice(),
00057                            &shader_module_create_info, nullptr, &shader_module);
00058
00059   ASSERT_VULKAN(result, "Failed to create a shader module!")
00060
00061   return shader_module;
00062 }
00063
00064 ShaderHelper::~ShaderHelper() {}
```

4.43 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ VulkanBuffer.cpp File Reference

```
#include "VulkanBuffer.h"
#include <stdexcept>
#include "MemoryHelper.h"
#include "Utilities.h"
```

Include dependency graph for VulkanBuffer.cpp:

4.44 VulkanBuffer.cpp

Go to the documentation of this file.

```
00001 #include "VulkanBuffer.h"
00002
00003 #include <stdexcept>
00004
00005 #include "MemoryHelper.h"
00006 #include "Utilities.h"
00007
00008 VulkanBuffer::VulkanBuffer() {}
00009
00010 void VulkanBuffer::create(VulkanDevice* device, VkDeviceSize buffer_size,
00011                           VkBufferUsageFlags buffer_usage_flags,
00012                           VkMemoryPropertyFlags buffer_propertiy_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_propertiy_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 }
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() {}
```

4.45 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ VulkanBufferManager.cpp File Reference

`#include "VulkanBufferManager.h"`

Include dependency graph for VulkanBufferManager.cpp:

4.46 VulkanBufferManager.cpp

Go to the documentation of this file.

```
00001 #include "VulkanBufferManager.h"
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) {
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 }
00027
00028 void VulkanBufferManager::copyImageBuffer(VkDevice device,
00029                                           VkQueue transfer_queue,
00030                                           VkCommandPool transfer_command_pool,
00031                                           VkBuffer src_buffer, VkImage image,
00032                                           uint32_t width, uint32_t height) {
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 }
00058
00059 VulkanBufferManager::~VulkanBufferManager() {}
```

4.47 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/VulkanDebug.cpp File Reference

```
#include "VulkanDebug.h"
#include "Utilities.h"
```

Include dependency graph for VulkanDebug.cpp:

Namespaces

- namespace debug

Functions

- VKAPI_ATTR VkBool32 VKAPI_CALL debug::debugUtilsMessengerCallback (VkDebugUtilsMessage↩ SeverityFlagBitsEXT messageSeverity, VkDebugUtilsMessageTypeFlagsEXT messageType, const Vk↩ DebugUtilsMessengerCallbackDataEXT ∗pCallbackData, void ∗pUserData)
- void debug::setupDebugging (VkInstance instance, VkDebugReportFlagsEXT flags, VkDebugReport↩ CallbackEXT callBack)
- void debug::freeDebugCallback (VkInstance instance)

Variables

- PFN_vkCreateDebugUtilsMessengerEXT debug::vkCreateDebugUtilsMessengerEXT
- PFN_vkDestroyDebugUtilsMessengerEXT debug::vkDestroyDebugUtilsMessengerEXT
- VkDebugUtilsMessengerEXT debug::debugUtilsMessenger

4.48 VulkanDebug.cpp

Go to the documentation of this file.

```
00001 #include "VulkanDebug.h"
00002
00003 #include "Utilities.h"
00004
00005 namespace debug {
00006 PFN_vkCreateDebugUtilsMessengerEXT vkCreateDebugUtilsMessengerEXT;
00007 PFN_vkDestroyDebugUtilsMessengerEXT vkDestroyDebugUtilsMessengerEXT;
00008 VkDebugUtilsMessengerEXT debugUtilsMessenger;
00009
00010 VKAPI_ATTR VkBool32 VKAPI_CALL debugUtilsMessengerCallback(
00011     VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
00012     VkDebugUtilsMessageTypeFlagsEXT messageType,
00013     const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
00014     void* pUserData) {
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 }
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
```

4.49 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ VulkanDevice.cpp File Reference

```
#include "VulkanDevice.h"
#include <string.h>
#include <set>
#include <string>
```

Include dependency graph for VulkanDevice.cpp:

4.50 VulkanDevice.cpp

Go to the documentation of this file.

```
00001 #include "VulkanDevice.h"
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 suppports 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_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 }
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 suppports 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 }
```

4.51 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ VulkanImage.cpp File Reference

```
#include "VulkanImage.h"
#include "MemoryHelper.h"
#include "Utilities.h"
```

Include dependency graph for VulkanImage.cpp:

4.52 VulkanImage.cpp

Go to the documentation of this file.

```
00001 #include "VulkanImage.h"
00002
00003 #include "MemoryHelper.h"
00004 #include "Utilities.h"
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 }
```

4.53 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ VulkanImageView.cpp File Reference

`#include "VulkanImageView.h"`

Include dependency graph for VulkanImageView.cpp:

4.54 VulkanImageView.cpp

Go to the documentation of this file.

```
00001 #include "VulkanImageView.h"
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() {}
```

4.55 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ VulkanInstance.cpp File Reference

```
#include "VulkanInstance.h"
#include <string.h>
#include <string>
```

Include dependency graph for VulkanInstance.cpp:

4.56 VulkanInstance.cpp

Go to the documentation of this file.

```
00001 #include "VulkanInstance.h"
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 \\__/";  // 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 creeate 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 }
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() {}
```

4.57 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/vulkan_base/↩ VulkanSwapChain.cpp File Reference

```
#include "VulkanSwapChain.h"
#include <limits>
#include "Utilities.h"
```

Include dependency graph for VulkanSwapChain.cpp:

4.58 VulkanSwapChain.cpp

Go to the documentation of this file.

```
00001 #include "VulkanSwapChain.h"
00002
00003 #include <limits>
00004
00005 #include "Utilities.h"
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   // tiple 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
00178   } else {
00179     int width, height;
00180     glfwGetFramebufferSize(window->get_window(), &width, &height);
00181
00182     // create new extent using window size
00183     VkExtent2D new_extent{};
00184     new_extent.width = static_cast<uint32_t>(width);
00185     new_extent.height = static_cast<uint32_t>(height);
00186
00187     // surface also defines max and min, so make sure within boundaries bly
00188     // clamping value
00189     new_extent.width = std::max(
00190         surface_capabilities.minImageExtent.width,
00191         std::min(surface_capabilities.maxImageExtent.width, new_extent.width));
00192     new_extent.height =
00193         std::max(surface_capabilities.minImageExtent.height,
00194                  std::min(surface_capabilities.maxImageExtent.height,
```

```
00195                           new_extent.height));
00196
00197     return new_extent;
00198   }
00199 }
```

4.59 C:/Users/jonas/Desktop/GraphicsEngineVulkan/Src/window/Window.cpp File Reference

```
#include "Window.h"
#include <imgui.h>
#include <imgui_impl_glfw.h>
#include <imgui_impl_vulkan.h>
#include <stdexcept>
```

Include dependency graph for Window.cpp:

Functions

- static void onErrorCallback (int error, const char *description)

4.59.1 Function Documentation

4.59.1.1 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 }
```

4.60 Window.cpp

Go to the documentation of this file.

```
00001 #include "Window.h"
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
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
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 occurance 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 substraction :)
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.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 }
00197
00198 Window::~Window() {}
```

Index