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.

```
                    {
  // Select prefix depending on flags passed to the callback
  std::string prefix("");

  if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT) {
    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"));
```

```
  vkDestroyDebugUtilsMessengerEXT =
      reinterpret_cast<PFN_vkDestroyDebugUtilsMessengerEXT>(
          vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT"));

  VkDebugUtilsMessengerCreateInfoEXT debugUtilsMessengerCI{};
  debugUtilsMessengerCI.sType =
      VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
  debugUtilsMessengerCI.messageSeverity =
      VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
      VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
  debugUtilsMessengerCI.messageType =
      VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
      VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT;
  debugUtilsMessengerCI.pfnUserCallback = debugUtilsMessengerCallback;
  ASSERT_VULKAN(vkCreateDebugUtilsMessengerEXT(instance, &debugUtilsMessengerCI,
                                               nullptr, &debugUtilsMessenger),
                "Failed to create debug messenger")
}
```

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,
     glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));*/

#else

// dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
// -50.0f));
#if SULO_MODE
  modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
#else
  modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
  modelMatrix = glm::rotate(modelMatrix, glm::radians(-90.f),
                            glm::vec3(1.0f, 0.0f, 0.0f));
  modelMatrix =
      glm::rotate(modelMatrix, glm::radians(90.f), glm::vec3(0.0f, 0.0f, 1.0f));
#endif

#endif

  return modelMatrix;
}
```

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()) {
    // poll all events incoming from user
    glfwPollEvents();

    // handle events for the camera
    camera->key_control(window->get_keys(), delta_time);
    camera->mouse_control(window->get_x_change(), window->get_y_change());

    float now = static_cast<float>(glfwGetTime());
    delta_time = now - last_time;
    last_time = now;

    scene->update_user_input(gui.get());

    vulkan_renderer.updateStateDueToUserInput(gui.get());
    vulkan_renderer.updateUniforms(scene.get(), camera.get(), window.get());

    //// retrieve updates from the UI
    gui->render();

    vulkan_renderer.drawFrame();
  }

  vulkan_renderer.finishAllRenderCommands();

  scene->cleanUp();
  gui->cleanUp();
  window->cleanUp();
  vulkan_renderer.cleanUp();

  return EXIT_SUCCESS;
}

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.

```
#include "GUI.h"

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

GUI::GUI(Window* window) { this->window = window; }

void GUI::initializeVulkanContext(VulkanDevice* device,
                                  const VkInstance& instance,
                                  const VkRenderPass& post_render_pass,
                                  const VkCommandPool& graphics_command_pool) {
  this->device = device;

  create_gui_context(window, instance, post_render_pass);
  create_fonts_and_upload(graphics_command_pool);
}

void GUI::render() {
  // Start the Dear ImGui frame
  ImGui_ImplVulkan_NewFrame();
  ImGui_ImplGlfw_NewFrame();
  ImGui::NewFrame();

  // ImGui::ShowDemoWindow();

  // render your GUI
  ImGui::Begin("GUI v1.4.4");

  if (ImGui::CollapsingHeader("Hot shader reload")) {
    if (ImGui::Button("All shader!")) {
      guiRendererSharedVars.shader_hot_reload_triggered = true;
    }
  }

  ImGui::Separator();

  static int e = 0;
  ImGui::RadioButton("Rasterizer", &e, 0);
  ImGui::SameLine();
  ImGui::RadioButton("Raytracing", &e, 1);
  ImGui::SameLine();
  ImGui::RadioButton("Path tracing", &e, 2);

  switch (e) {
    case 0:
      guiRendererSharedVars.raytracing = false;
      guiRendererSharedVars.pathTracing = false;
      break;
    case 1:
      guiRendererSharedVars.raytracing = true;
      guiRendererSharedVars.pathTracing = false;
      break;
    case 2:
      guiRendererSharedVars.raytracing = false;
      guiRendererSharedVars.pathTracing = true;
      break;
  }
  // ImGui::Checkbox("Ray tracing", &guiRendererSharedVars.raytracing);

  ImGui::Separator();

  if (ImGui::CollapsingHeader("Graphic Settings")) {
    if (ImGui::TreeNode("Directional Light")) {
      ImGui::Separator();
      ImGui::SliderFloat("Ambient intensity",
                         &guiSceneSharedVars.direcional_light_radiance, 0.0f,
                         50.0f);
      ImGui::Separator();
      // Edit a color (stored as ~4 floats)
      ImGui::ColorEdit3("Directional Light Color",
                        guiSceneSharedVars.directional_light_color);
      ImGui::Separator();
      ImGui::SliderFloat3("Light Direction",
                          guiSceneSharedVars.directional_light_direction, -1.f,
                          1.0f);

      ImGui::TreePop();
    }
  }

  ImGui::Separator();

  if (ImGui::CollapsingHeader("GUI Settings")) {
    ImGuiStyle& style = ImGui::GetStyle();

    if (ImGui::SliderFloat("Frame Rounding", &style.FrameRounding, 0.0f, 12.0f,
                           "%.0f")) {
      style.GrabRounding = style.FrameRounding;  // Make GrabRounding always the
                                                 // same value as FrameRounding
    }
    {
      bool border = (style.FrameBorderSize > 0.0f);
      if (ImGui::Checkbox("FrameBorder", &border)) {
        style.FrameBorderSize = border ? 1.0f : 0.0f;
      }
    }
    ImGui::SliderFloat("WindowRounding", &style.WindowRounding, 0.0f, 12.0f,
                       "%.0f");
  }

  ImGui::Separator();

  if (ImGui::CollapsingHeader("KEY Bindings")) {
    ImGui::Text(
        "WASD for moving Forward, backward and to the side\n QE for rotating ");
  }

  ImGui::Separator();

  ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
              1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);

  ImGui::End();
}

void GUI::cleanUp() {
  // clean up of GUI stuff
  ImGui_ImplVulkan_Shutdown();
  ImGui_ImplGlfw_Shutdown();
  ImGui::DestroyContext();
  vkDestroyDescriptorPool(device->getLogicalDevice(), gui_descriptor_pool,
                          nullptr);
}

void GUI::create_gui_context(Window* window, const VkInstance& instance,
                             const VkRenderPass& post_render_pass) {
  IMGUI_CHECKVERSION();
  ImGui::CreateContext();
  ImGuiIO& io = ImGui::GetIO();
  (void)io;

  float size_pixels = 18;

  std::stringstream fontDir;
  fontDir « CMAKELISTS_DIR;
  fontDir « "/ExternalLib/IMGUI/misc/fonts/";

  std::stringstream robo_font;
  robo_font « fontDir.str() « "Roboto-Medium.ttf";
  std::stringstream Cousine_font;
  Cousine_font « fontDir.str() « "Cousine-Regular.ttf";
  std::stringstream DroidSans_font;
  DroidSans_font « fontDir.str() « "DroidSans.ttf";
  std::stringstream Karla_font;
  Karla_font « fontDir.str() « "Karla-Regular.ttf";
  std::stringstream proggy_clean_font;
  proggy_clean_font « fontDir.str() « "ProggyClean.ttf";
  std::stringstream proggy_tiny_font;
  proggy_tiny_font « fontDir.str() « "ProggyTiny.ttf";

  io.Fonts->AddFontFromFileTTF(robo_font.str().c_str(), size_pixels);
  io.Fonts->AddFontFromFileTTF(Cousine_font.str().c_str(), size_pixels);
  io.Fonts->AddFontFromFileTTF(DroidSans_font.str().c_str(), size_pixels);
  io.Fonts->AddFontFromFileTTF(Karla_font.str().c_str(), size_pixels);
  io.Fonts->AddFontFromFileTTF(proggy_clean_font.str().c_str(), size_pixels);
  io.Fonts->AddFontFromFileTTF(proggy_tiny_font.str().c_str(), size_pixels);

  ImGui::PushStyleVar(ImGuiStyleVar_WindowRounding, 10);
  ImGui::PushStyleVar(ImGuiStyleVar_FrameRounding, 10);
  ImGui::PushStyleVar(ImGuiStyleVar_FrameBorderSize, 1);
  io.ConfigFlags |=
      ImGuiConfigFlags_NavEnableKeyboard;  // Enable Keyboard Controls
  io.ConfigFlags |= ImGuiConfigFlags_NavEnableSetMousePos;
  io.WantCaptureMouse = true;
  // io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad;      // Enable Gamepad
  // Controls

  // Setup Dear ImGui style
  ImGui::StyleColorsDark();
  // ImGui::StyleColorsClassic();

  ImGui_ImplGlfw_InitForVulkan(window->get_window(), false);

  // Create Descriptor Pool
  VkDescriptorPoolSize gui_pool_sizes[] = {
      {VK_DESCRIPTOR_TYPE_SAMPLER, 10},
      {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 10},
      {VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 10},
      {VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 10},
      {VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, 10},
      {VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, 10},
      {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 10},
      {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 10},
      {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 10},
      {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, 10},
      {VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, 100}};

  VkDescriptorPoolCreateInfo gui_pool_info = {};
  gui_pool_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
  gui_pool_info.flags = VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT;
  gui_pool_info.maxSets = 10 * IM_ARRAYSIZE(gui_pool_sizes);
  gui_pool_info.poolSizeCount = (uint32_t)IM_ARRAYSIZE(gui_pool_sizes);
  gui_pool_info.pPoolSizes = gui_pool_sizes;

  VkResult result =
      vkCreateDescriptorPool(device->getLogicalDevice(), &gui_pool_info,
                             nullptr, &gui_descriptor_pool);
  ASSERT_VULKAN(result, "Failed to create a gui descriptor pool!")

  QueueFamilyIndices indices = device->getQueueFamilies();

  ImGui_ImplVulkan_InitInfo init_info = {};
  init_info.Instance = instance;
  init_info.PhysicalDevice = device->getPhysicalDevice();
  init_info.Device = device->getLogicalDevice();
  init_info.QueueFamily = indices.graphics_family;
  init_info.Queue = device->getGraphicsQueue();
  init_info.DescriptorPool = gui_descriptor_pool;
  init_info.PipelineCache = VK_NULL_HANDLE;
  init_info.MinImageCount = 2;
  init_info.ImageCount = MAX_FRAME_DRAWS;
  init_info.Allocator = VK_NULL_HANDLE;
  init_info.CheckVkResultFn = VK_NULL_HANDLE;
  init_info.Subpass = 0;
  init_info.MSAASamples = VK_SAMPLE_COUNT_1_BIT;

  ImGui_ImplVulkan_Init(&init_info, post_render_pass);
}

void GUI::create_fonts_and_upload(const VkCommandPool& graphics_command_pool) {
  VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
      device->getLogicalDevice(), graphics_command_pool);
  ImGui_ImplVulkan_CreateFontsTexture(command_buffer);
  commandBufferManager.endAndSubmitCommandBuffer(
      device->getLogicalDevice(), graphics_command_pool,
      device->getGraphicsQueue(), command_buffer);

  // wait until no actions being run on device before destroying
  vkDeviceWaitIdle(device->getLogicalDevice());
  // clear font textures from cpu data
  ImGui_ImplVulkan_DestroyFontUploadObjects();
}

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.

```
#include "App.h"

int main() {
  App application;
  return application.run();
}
```

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.

```
#include "Allocator.h"

#include "Utilities.h"

Allocator::Allocator() {}

Allocator::Allocator(const VkDevice& device,
                     const VkPhysicalDevice& physicalDevice,
                     const VkInstance& instance) {
  // see here:
  // https://gpuopen-librariesandsdks.github.io/VulkanMemoryAllocator/html/quick_start.html
  VmaAllocatorCreateInfo allocatorCreateInfo = {};
  allocatorCreateInfo.flags = VMA_ALLOCATOR_CREATE_BUFFER_DEVICE_ADDRESS_BIT;
  allocatorCreateInfo.vulkanApiVersion = VK_API_VERSION_1_3;
  allocatorCreateInfo.physicalDevice = physicalDevice;
  allocatorCreateInfo.device = device;
  allocatorCreateInfo.instance = instance;

  ASSERT_VULKAN(vmaCreateAllocator(&allocatorCreateInfo, &vmaAllocator),
                "Failed to create vma allocator!")
}

void Allocator::cleanUp() { vmaDestroyAllocator(vmaAllocator); }

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.

```
#include "ASManager.h"

ASManager::ASManager() {}

void ASManager::createASForScene(VulkanDevice* device,
                                 VkCommandPool commandPool, Scene* scene) {
  this->vulkanDevice = device;
  createBLAS(device, commandPool, scene);
  createTLAS(device, commandPool, scene);
}

void ASManager::createBLAS(VulkanDevice* device, VkCommandPool commandPool,
                           Scene* scene) {
  // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
  // all functionality from extensions has to be loaded in the beginning
  // we need a reference to the device location of our geometry laying on the
  // graphics card we already uploaded objects and created vertex and index
  // buffers respectively

  PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
      (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkGetBufferDeviceAddress");

  std::vector<BlasInput> blas_input(scene->getModelCount());

  for (uint32_t model_index = 0;
       model_index < static_cast<uint32_t>(scene->getModelCount());
       model_index++) {
    std::shared_ptr<Model> mesh_model = scene->get_model_list()[model_index];
    // blas_input.emplace_back();
    blas_input[model_index].as_geometry.reserve(mesh_model->getMeshCount());
    blas_input[model_index].as_build_offset_info.reserve(
        mesh_model->getMeshCount());

    for (size_t mesh_index = 0; mesh_index < mesh_model->getMeshCount();
         mesh_index++) {
      VkAccelerationStructureGeometryKHR acceleration_structure_geometry{};
      VkAccelerationStructureBuildRangeInfoKHR
          acceleration_structure_build_range_info{};

      objectToVkGeometryKHR(device, mesh_model->getMesh(mesh_index),
                            acceleration_structure_geometry,
                            acceleration_structure_build_range_info);
      // this only specifies the acceleration structure
      // we are building it in the end for the whole model with the build
      // command

      blas_input[model_index].as_geometry.push_back(
          acceleration_structure_geometry);
      blas_input[model_index].as_build_offset_info.push_back(
          acceleration_structure_build_range_info);
    }
  }

  std::vector<BuildAccelerationStructure> build_as_structures;
  build_as_structures.resize(scene->getModelCount());

  VkDeviceSize max_scratch_size = 0;
  VkDeviceSize total_size_all_BLAS = 0;

  for (unsigned int i = 0; i < scene->getModelCount(); i++) {
    VkDeviceSize current_scretch_size = 0;
    VkDeviceSize current_size = 0;

    createAccelerationStructureInfosBLAS(device, build_as_structures[i],
                                         blas_input[i], current_scretch_size,
                                         current_size);

    total_size_all_BLAS += current_size;
    max_scratch_size = std::max(max_scratch_size, current_scretch_size);
  }

  VulkanBuffer scratchBuffer;

  scratchBuffer.create(device, max_scratch_size,
                       VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
                           VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
                           VK_BUFFER_USAGE_TRANSFER_DST_BIT,
                       VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
                           VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

  VkBufferDeviceAddressInfo scratch_buffer_device_address_info{};
  scratch_buffer_device_address_info.sType =
      VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
  scratch_buffer_device_address_info.buffer = scratchBuffer.getBuffer();

  VkDeviceAddress scratch_buffer_address = pvkGetBufferDeviceAddressKHR(
      device->getLogicalDevice(), &scratch_buffer_device_address_info);

  VkDeviceOrHostAddressKHR scratch_device_or_host_address{};
  scratch_device_or_host_address.deviceAddress = scratch_buffer_address;

  VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
      device->getLogicalDevice(), commandPool);

  for (size_t i = 0; i < scene->getModelCount(); i++) {
    createSingleBlas(device, command_buffer, build_as_structures[i],
                     scratch_buffer_address);

    VkMemoryBarrier barrier;
    barrier.pNext = nullptr;
    barrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
    barrier.srcAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
    barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR;

    vkCmdPipelineBarrier(command_buffer,
                         VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
                         VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
                         0, 1, &barrier, 0, nullptr, 0, nullptr);
  }

  commandBufferManager.endAndSubmitCommandBuffer(
      device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
      command_buffer);

  for (auto& b : build_as_structures) {
    blas.emplace_back(b.single_blas);
  }

  scratchBuffer.cleanUp();
}

void ASManager::createTLAS(VulkanDevice* device, VkCommandPool commandPool,
                           Scene* scene) {
  // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
  // all functionality from extensions has to be loaded in the beginning
  // we need a reference to the device location of our geometry laying on the
  // graphics card we already uploaded objects and created vertex and index
  // buffers respectively
  PFN_vkGetAccelerationStructureBuildSizesKHR
      pvkGetAccelerationStructureBuildSizesKHR =
          (PFN_vkGetAccelerationStructureBuildSizesKHR)vkGetDeviceProcAddr(
              device->getLogicalDevice(),
              "vkGetAccelerationStructureBuildSizesKHR");

  PFN_vkCreateAccelerationStructureKHR pvkCreateAccelerationStructureKHR =
      (PFN_vkCreateAccelerationStructureKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkCreateAccelerationStructureKHR");

  PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
      (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkGetBufferDeviceAddress");

  PFN_vkCmdBuildAccelerationStructuresKHR pvkCmdBuildAccelerationStructuresKHR =
      (PFN_vkCmdBuildAccelerationStructuresKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkCmdBuildAccelerationStructuresKHR");

  PFN_vkGetAccelerationStructureDeviceAddressKHR
      pvkGetAccelerationStructureDeviceAddressKHR =
          (PFN_vkGetAccelerationStructureDeviceAddressKHR)vkGetDeviceProcAddr(
              device->getLogicalDevice(),
              "vkGetAccelerationStructureDeviceAddressKHR");

  std::vector<VkAccelerationStructureInstanceKHR> tlas_instances;
  tlas_instances.reserve(scene->getModelCount());

  for (size_t model_index = 0; model_index < scene->getModelCount();
       model_index++) {
    // glm uses column major matrices so transpose it for Vulkan want row major
    // here
    glm::mat4 transpose_transform =
        glm::transpose(scene->getModelMatrix(static_cast<int>(model_index)));
    VkTransformMatrixKHR out_matrix;
    memcpy(&out_matrix, &transpose_transform, sizeof(VkTransformMatrixKHR));

    VkAccelerationStructureDeviceAddressInfoKHR
        acceleration_structure_device_address_info{};
    acceleration_structure_device_address_info.sType =
        VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_DEVICE_ADDRESS_INFO_KHR;
    acceleration_structure_device_address_info.accelerationStructure =
        blas[model_index].vulkanAS;

    VkDeviceAddress acceleration_structure_device_address =
        pvkGetAccelerationStructureDeviceAddressKHR(
            device->getLogicalDevice(),
            &acceleration_structure_device_address_info);

    VkAccelerationStructureInstanceKHR geometry_instance{};
    geometry_instance.transform = out_matrix;
    geometry_instance.instanceCustomIndex =
        model_index;  // gl_InstanceCustomIndexEXT
    geometry_instance.mask = 0xFF;
    geometry_instance.instanceShaderBindingTableRecordOffset = 0;
    geometry_instance.flags =
        VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
    geometry_instance.accelerationStructureReference =
        acceleration_structure_device_address;
    geometry_instance.instanceShaderBindingTableRecordOffset =
        0;  // same hit group for all objects

    tlas_instances.emplace_back(geometry_instance);
  }

  VkCommandBuffer command_buffer = commandBufferManager.beginCommandBuffer(
      device->getLogicalDevice(), commandPool);

  VulkanBuffer geometryInstanceBuffer;

  vulkanBufferManager.createBufferAndUploadVectorOnDevice(
      device, commandPool, geometryInstanceBuffer,
      VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
          VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
          VK_BUFFER_USAGE_TRANSFER_DST_BIT,
      VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
          VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
      tlas_instances);

  VkBufferDeviceAddressInfo geometry_instance_buffer_device_address_info{};
  geometry_instance_buffer_device_address_info.sType =
      VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
  geometry_instance_buffer_device_address_info.buffer =
      geometryInstanceBuffer.getBuffer();

  VkDeviceAddress geometry_instance_buffer_address =
      pvkGetBufferDeviceAddressKHR(
          device->getLogicalDevice(),
          &geometry_instance_buffer_device_address_info);

  // Make sure the copy of the instance buffer are copied before triggering the
  // acceleration structure build
  VkMemoryBarrier barrier;
  barrier.pNext = nullptr;
  barrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
  barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
  vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
                       VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
                       0, 1, &barrier, 0, nullptr, 0, nullptr);

  VkAccelerationStructureGeometryInstancesDataKHR
      acceleration_structure_geometry_instances_data{};
  acceleration_structure_geometry_instances_data.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR;
  acceleration_structure_geometry_instances_data.pNext = nullptr;
  acceleration_structure_geometry_instances_data.data.deviceAddress =
      geometry_instance_buffer_address;

  VkAccelerationStructureGeometryKHR topAS_acceleration_structure_geometry{};
  topAS_acceleration_structure_geometry.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
  topAS_acceleration_structure_geometry.pNext = nullptr;
  topAS_acceleration_structure_geometry.geometryType =
      VK_GEOMETRY_TYPE_INSTANCES_KHR;
  topAS_acceleration_structure_geometry.geometry.instances =
      acceleration_structure_geometry_instances_data;

  // find sizes
  VkAccelerationStructureBuildGeometryInfoKHR
      acceleration_structure_build_geometry_info{};
  acceleration_structure_build_geometry_info.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
  acceleration_structure_build_geometry_info.pNext = nullptr;
  acceleration_structure_build_geometry_info.type =
      VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
  acceleration_structure_build_geometry_info.flags =
      VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
```

```
acceleration_structure_build_geometry_info.mode =
    VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
acceleration_structure_build_geometry_info.srcAccelerationStructure =
    VK_NULL_HANDLE;
acceleration_structure_build_geometry_info.geometryCount = 1;
acceleration_structure_build_geometry_info.pGeometries =
    &topAS_acceleration_structure_geometry;

VkAccelerationStructureBuildSizesInfoKHR
    acceleration_structure_build_sizes_info{};
acceleration_structure_build_sizes_info.sType =
    VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR;
acceleration_structure_build_sizes_info.pNext = nullptr;
acceleration_structure_build_sizes_info.accelerationStructureSize = 0;
acceleration_structure_build_sizes_info.updateScratchSize = 0;
acceleration_structure_build_sizes_info.buildScratchSize = 0;

uint32_t count_instance = static_cast<uint32_t>(tlas_instances.size());
pvkGetAccelerationStructureBuildSizesKHR(
    device->getLogicalDevice(), VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_KHR,
    &acceleration_structure_build_geometry_info, &count_instance,
    &acceleration_structure_build_sizes_info);

// now we got the sizes
VulkanBuffer& tlasVulkanBuffer = tlas.vulkanBuffer;
tlasVulkanBuffer.create(
    device, acceleration_structure_build_sizes_info.accelerationStructureSize,
    VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR |
        VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
        VK_BUFFER_USAGE_TRANSFER_DST_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
        VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT);

VkAccelerationStructureCreateInfoKHR acceleration_structure_create_info{};
acceleration_structure_create_info.sType =
    VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
acceleration_structure_create_info.pNext = nullptr;
acceleration_structure_create_info.createFlags = 0;
acceleration_structure_create_info.buffer = tlasVulkanBuffer.getBuffer();
acceleration_structure_create_info.offset = 0;
acceleration_structure_create_info.size =
    acceleration_structure_build_sizes_info.accelerationStructureSize;
acceleration_structure_create_info.type =
    VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
acceleration_structure_create_info.deviceAddress = 0;

VkAccelerationStructureKHR& tlAS = tlas.vulkanAS;
pvkCreateAccelerationStructureKHR(device->getLogicalDevice(),
                                  &acceleration_structure_create_info,
                                  nullptr, &tlAS);

VulkanBuffer scratchBuffer;

scratchBuffer.create(device,
                     acceleration_structure_build_sizes_info.buildScratchSize,
                     VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
                         VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
                         VK_BUFFER_USAGE_TRANSFER_DST_BIT,
                     VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
                         VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT);

VkBufferDeviceAddressInfo scratch_buffer_device_address_info{};
scratch_buffer_device_address_info.sType =
    VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
scratch_buffer_device_address_info.buffer = scratchBuffer.getBuffer();

VkDeviceAddress scratch_buffer_address = pvkGetBufferDeviceAddressKHR(
    device->getLogicalDevice(), &scratch_buffer_device_address_info);

// update build info
acceleration_structure_build_geometry_info.scratchData.deviceAddress =
    scratch_buffer_address;
acceleration_structure_build_geometry_info.srcAccelerationStructure =
    VK_NULL_HANDLE;
acceleration_structure_build_geometry_info.dstAccelerationStructure = tlAS;

VkAccelerationStructureBuildRangeInfoKHR
    acceleration_structure_build_range_info{};
acceleration_structure_build_range_info.primitiveCount =
    scene->getModelCount();
acceleration_structure_build_range_info.primitiveOffset = 0;
acceleration_structure_build_range_info.firstVertex = 0;
acceleration_structure_build_range_info.transformOffset = 0;

VkAccelerationStructureBuildRangeInfoKHR*
    acceleration_structure_build_range_infos =
        &acceleration_structure_build_range_info;
```

```

  pvkCmdBuildAccelerationStructuresKHR(
      command_buffer, 1, &acceleration_structure_build_geometry_info,
      &acceleration_structure_build_range_infos);

  commandBufferManager.endAndSubmitCommandBuffer(
      device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
      command_buffer);
  scratchBuffer.cleanUp();
  geometryInstanceBuffer.cleanUp();
}

void ASManager::cleanUp() {
  PFN_vkDestroyAccelerationStructureKHR pvkDestroyAccelerationStructureKHR =
      (PFN_vkDestroyAccelerationStructureKHR)vkGetDeviceProcAddr(
          vulkanDevice->getLogicalDevice(),
          "vkDestroyAccelerationStructureKHR");

  pvkDestroyAccelerationStructureKHR(vulkanDevice->getLogicalDevice(),
                                     tlas.vulkanAS, nullptr);

  tlas.vulkanBuffer.cleanUp();

  for (size_t index = 0; index < blas.size(); index++) {
    pvkDestroyAccelerationStructureKHR(vulkanDevice->getLogicalDevice(),
                                       blas[index].vulkanAS, nullptr);

    blas[index].vulkanBuffer.cleanUp();
  }
}

ASManager::~ASManager() {}

void ASManager::createSingleBlas(
    VulkanDevice* device, VkCommandBuffer command_buffer,
    BuildAccelerationStructure& build_as_structure,
    VkDeviceAddress scratch_device_or_host_address) {
  PFN_vkCreateAccelerationStructureKHR pvkCreateAccelerationStructureKHR =
      (PFN_vkCreateAccelerationStructureKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkCreateAccelerationStructureKHR");

  PFN_vkCmdBuildAccelerationStructuresKHR pvkCmdBuildAccelerationStructuresKHR =
      (PFN_vkCmdBuildAccelerationStructuresKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkCmdBuildAccelerationStructuresKHR");

  VkAccelerationStructureCreateInfoKHR acceleration_structure_create_info{};
  acceleration_structure_create_info.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
  acceleration_structure_create_info.type =
      VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
  acceleration_structure_create_info.size =
      build_as_structure.size_info.accelerationStructureSize;
  VulkanBuffer& blasVulkanBuffer = build_as_structure.single_blas.vulkanBuffer;
  blasVulkanBuffer.create(
      device, build_as_structure.size_info.accelerationStructureSize,
      VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR |
          VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
          VK_BUFFER_USAGE_TRANSFER_DST_BIT,
      VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT |
          VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

  acceleration_structure_create_info.buffer = blasVulkanBuffer.getBuffer();
  VkAccelerationStructureKHR& blas_as = build_as_structure.single_blas.vulkanAS;
  pvkCreateAccelerationStructureKHR(device->getLogicalDevice(),
                                    &acceleration_structure_create_info,
                                    nullptr, &blas_as);

  build_as_structure.build_info.dstAccelerationStructure = blas_as;
  build_as_structure.build_info.scratchData.deviceAddress =
      scratch_device_or_host_address;

  pvkCmdBuildAccelerationStructuresKHR(command_buffer, 1,
                                       &build_as_structure.build_info,
                                       &build_as_structure.range_info);
}

void ASManager::createAccelerationStructureInfosBLAS(
    VulkanDevice* device, BuildAccelerationStructure& build_as_structure,
    BlasInput& blas_input, VkDeviceSize& current_scretch_size,
    VkDeviceSize& current_size) {
  PFN_vkGetAccelerationStructureBuildSizesKHR
      pvkGetAccelerationStructureBuildSizesKHR =
          (PFN_vkGetAccelerationStructureBuildSizesKHR)vkGetDeviceProcAddr(
              device->getLogicalDevice(),
              "vkGetAccelerationStructureBuildSizesKHR");

  build_as_structure.build_info.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
  build_as_structure.build_info.type =
      VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
  build_as_structure.build_info.flags =
      VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
  build_as_structure.build_info.mode =
      VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
  build_as_structure.build_info.geometryCount =
      static_cast<uint32_t>(blas_input.as_geometry.size());
  build_as_structure.build_info.pGeometries = blas_input.as_geometry.data();

  build_as_structure.range_info = blas_input.as_build_offset_info.data();

  build_as_structure.size_info.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR;

  std::vector<uint32_t> max_primitive_cnt(
      blas_input.as_build_offset_info.size());

  for (uint32_t temp = 0;
       temp < static_cast<uint32_t>(blas_input.as_build_offset_info.size());
       temp++)
    max_primitive_cnt[temp] =
        blas_input.as_build_offset_info[temp].primitiveCount;

  pvkGetAccelerationStructureBuildSizesKHR(
      device->getLogicalDevice(),
      VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR,
      &build_as_structure.build_info, max_primitive_cnt.data(),
      &build_as_structure.size_info);

  current_size = build_as_structure.size_info.accelerationStructureSize;
  current_scretch_size = build_as_structure.size_info.buildScratchSize;
}

void ASManager::objectToVkGeometryKHR(
    VulkanDevice* device, Mesh* mesh,
    VkAccelerationStructureGeometryKHR& acceleration_structure_geometry,
    VkAccelerationStructureBuildRangeInfoKHR&
        acceleration_structure_build_range_info) {
  // LOAD ALL NECESSARY FUNCTIONS STRAIGHT IN THE BEGINNING
  // all functionality from extensions has to be loaded in the beginning
  // we need a reference to the device location of our geometry laying on the
  // graphics card we already uploaded objects and created vertex and index
  // buffers respectively
  PFN_vkGetBufferDeviceAddressKHR pvkGetBufferDeviceAddressKHR =
      (PFN_vkGetBufferDeviceAddressKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkGetBufferDeviceAddress");

  // all starts with the address of our vertex and index data we already
  // uploaded in buffers earlier when loading the meshes/models
  VkBufferDeviceAddressInfo vertex_buffer_device_address_info{};
  vertex_buffer_device_address_info.sType =
      VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
  vertex_buffer_device_address_info.buffer = mesh->getVertexBuffer();
  vertex_buffer_device_address_info.pNext = nullptr;

  VkBufferDeviceAddressInfo index_buffer_device_address_info{};
  index_buffer_device_address_info.sType =
      VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
  index_buffer_device_address_info.buffer = mesh->getIndexBuffer();
  index_buffer_device_address_info.pNext = nullptr;

  // receiving address to move on
  VkDeviceAddress vertex_buffer_address = pvkGetBufferDeviceAddressKHR(
      device->getLogicalDevice(), &vertex_buffer_device_address_info);
  VkDeviceAddress index_buffer_address = pvkGetBufferDeviceAddressKHR(
      device->getLogicalDevice(), &index_buffer_device_address_info);

  // convert to const address for further processing
  VkDeviceOrHostAddressConstKHR vertex_device_or_host_address_const{};
  vertex_device_or_host_address_const.deviceAddress = vertex_buffer_address;

  VkDeviceOrHostAddressConstKHR index_device_or_host_address_const{};
  index_device_or_host_address_const.deviceAddress = index_buffer_address;

  VkAccelerationStructureGeometryTrianglesDataKHR
      acceleration_structure_triangles_data{};
  acceleration_structure_triangles_data.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR;
  acceleration_structure_triangles_data.pNext = nullptr;
  acceleration_structure_triangles_data.vertexFormat =
      VK_FORMAT_R32G32B32_SFLOAT;
  acceleration_structure_triangles_data.vertexData =
      vertex_device_or_host_address_const;
  acceleration_structure_triangles_data.vertexStride = sizeof(Vertex);
  acceleration_structure_triangles_data.maxVertex = mesh->getVertexCount();
  acceleration_structure_triangles_data.indexType = VK_INDEX_TYPE_UINT32;
  acceleration_structure_triangles_data.indexData =
      index_device_or_host_address_const;

  // can also be instances or AABBs; not covered here
  // but to identify as triangles put it ito these struct
  VkAccelerationStructureGeometryDataKHR acceleration_structure_geometry_data{};
  acceleration_structure_geometry_data.triangles =
      acceleration_structure_triangles_data;

  acceleration_structure_geometry.sType =
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
  acceleration_structure_geometry.pNext = nullptr;
  acceleration_structure_geometry.geometryType = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
  acceleration_structure_geometry.geometry =
      acceleration_structure_geometry_data;
  acceleration_structure_geometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR;

  // we have triangles so divide the number of vertices with 3!!
  // for our simple case a no brainer
  // take entire data to build BLAS
  // number of indices is truly the stick point here
  acceleration_structure_build_range_info.primitiveCount =
      mesh->getIndexCount() / 3;
  acceleration_structure_build_range_info.primitiveOffset = 0;
  acceleration_structure_build_range_info.firstVertex = 0;
  acceleration_structure_build_range_info.transformOffset = 0;
}
```

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.

```
#include "CommandBufferManager.h"

CommandBufferManager::CommandBufferManager() {}

VkCommandBuffer CommandBufferManager::beginCommandBuffer(
    VkDevice device, VkCommandPool command_pool) {
  // command buffer to hold transfer commands
  VkCommandBuffer command_buffer;

  // command buffer details
  VkCommandBufferAllocateInfo alloc_info{};
  alloc_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
  alloc_info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
  alloc_info.commandPool = command_pool;
  alloc_info.commandBufferCount = 1;

  // allocate command buffer from pool
  vkAllocateCommandBuffers(device, &alloc_info, &command_buffer);

  // infromation to begin the command buffer record
  VkCommandBufferBeginInfo begin_info{};
  begin_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
  // we are only using the command buffer once, so set up for one time submit
  begin_info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

  // begin recording transfer commands
  vkBeginCommandBuffer(command_buffer, &begin_info);

  return command_buffer;
}

void CommandBufferManager::endAndSubmitCommandBuffer(
    VkDevice device, VkCommandPool command_pool, VkQueue queue,
    VkCommandBuffer& command_buffer) {
  // end commands
  VkResult result = vkEndCommandBuffer(command_buffer);
  ASSERT_VULKAN(result, "Failed to end command buffer!")

  // queue submission information
  VkSubmitInfo submit_info{};
  submit_info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  submit_info.commandBufferCount = 1;
  submit_info.pCommandBuffers = &command_buffer;

  // submit transfer command to transfer queue and wait until it finishes
  result = vkQueueSubmit(queue, 1, &submit_info, VK_NULL_HANDLE);
  ASSERT_VULKAN(result, "Failed to submit to queue!")

  result = vkQueueWaitIdle(queue);
  ASSERT_VULKAN(result, "Failed to wait Idle!")

  // free temporary command buffer back to pool
  vkFreeCommandBuffers(device, command_pool, 1, &command_buffer);
}

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.

```
#include "PathTracing.h"

#include <algorithm>
#include <array>

#include "File.h"
#include "ShaderHelper.h"

// Good source:
// https://github.com/nvpro-samples/vk_mini_path_tracer/blob/main/vk_mini_path_tracer/main.cpp

PathTracing::PathTracing() {}

void PathTracing::init(
    VulkanDevice* device,
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
  this->device = device;

  VkPhysicalDeviceProperties physicalDeviceProps =
      device->getPhysicalDeviceProperties();
  timeStampPeriod = physicalDeviceProps.limits.timestampPeriod;

  // save the limits for handling all special cases later on
  computeLimits.maxComputeWorkGroupCount[0] =
      physicalDeviceProps.limits.maxComputeWorkGroupCount[0];
  computeLimits.maxComputeWorkGroupCount[1] =
      physicalDeviceProps.limits.maxComputeWorkGroupCount[1];
  computeLimits.maxComputeWorkGroupCount[2] =
      physicalDeviceProps.limits.maxComputeWorkGroupCount[2];

  computeLimits.maxComputeWorkGroupInvocations =
      physicalDeviceProps.limits.maxComputeWorkGroupInvocations;

  computeLimits.maxComputeWorkGroupSize[0] =
      physicalDeviceProps.limits.maxComputeWorkGroupSize[0];
  computeLimits.maxComputeWorkGroupSize[1] =
      physicalDeviceProps.limits.maxComputeWorkGroupSize[1];
  computeLimits.maxComputeWorkGroupSize[2] =
      physicalDeviceProps.limits.maxComputeWorkGroupSize[2];

  queryResults.resize(query_count);
  createQueryPool();

  createPipeline(descriptorSetLayouts);
}

void PathTracing::shaderHotReload(
    const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
  vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
  createPipeline(descriptor_set_layouts);
}

void PathTracing::recordCommands(
    VkCommandBuffer& commandBuffer, uint32_t image_index,
    VulkanImage& vulkanImage, VulkanSwapChain* vulkanSwapChain,
    const std::vector<VkDescriptorSet>& descriptorSets) {
  // we have reset the pool; hence start by 0
  uint32_t query = 0;

  vkCmdResetQueryPool(commandBuffer, queryPool, 0, query_count);

  vkCmdWriteTimestamp(
      commandBuffer,
      VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,
      query++);

  QueueFamilyIndices indices = device->getQueueFamilies();

  VkImageSubresourceRange subresourceRange{};
  subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
  subresourceRange.baseMipLevel = 0;
  subresourceRange.baseArrayLayer = 0;
  subresourceRange.levelCount = 1;
  subresourceRange.layerCount = 1;

  VkImageMemoryBarrier presentToPathTracingImageBarrier{};
  presentToPathTracingImageBarrier.sType =
      VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
  presentToPathTracingImageBarrier.pNext = nullptr;
  presentToPathTracingImageBarrier.srcQueueFamilyIndex =
      indices.graphics_family;
  presentToPathTracingImageBarrier.dstQueueFamilyIndex = indices.compute_family;
  presentToPathTracingImageBarrier.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
  presentToPathTracingImageBarrier.dstAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
  presentToPathTracingImageBarrier.oldLayout = VK_IMAGE_LAYOUT_GENERAL;
  presentToPathTracingImageBarrier.newLayout = VK_IMAGE_LAYOUT_GENERAL;
  presentToPathTracingImageBarrier.subresourceRange = subresourceRange;
  presentToPathTracingImageBarrier.image = vulkanImage.getImage();

  vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,
                       VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,

                       0, 0, nullptr, 0, nullptr, 1,
                       &presentToPathTracingImageBarrier);

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

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

  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);

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

  uint32_t workGroupCountX =
      std::max((imageSize.width + specializationData.specWorkGroupSizeX - 1) /
                   specializationData.specWorkGroupSizeX,
               1U);
  uint32_t workGroupCountY =
      std::max((imageSize.height + specializationData.specWorkGroupSizeY - 1) /
                   specializationData.specWorkGroupSizeY,
               1U);
  uint32_t workGroupCountZ = 1;

  vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY,
                workGroupCountZ);
  VkImageMemoryBarrier pathTracingToPresentImageBarrier{};
  pathTracingToPresentImageBarrier.sType =
      VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
  pathTracingToPresentImageBarrier.pNext = nullptr;
  pathTracingToPresentImageBarrier.srcQueueFamilyIndex = indices.compute_family;
  pathTracingToPresentImageBarrier.dstQueueFamilyIndex =
      indices.graphics_family;
  pathTracingToPresentImageBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
  pathTracingToPresentImageBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
  pathTracingToPresentImageBarrier.oldLayout = VK_IMAGE_LAYOUT_GENERAL;
  pathTracingToPresentImageBarrier.newLayout = VK_IMAGE_LAYOUT_GENERAL;
  pathTracingToPresentImageBarrier.image = vulkanImage.getImage();
  pathTracingToPresentImageBarrier.subresourceRange = subresourceRange;

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

  vkCmdWriteTimestamp(
      commandBuffer,
      VkPipelineStageFlagBits::VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, queryPool,
      query++);
  VkResult result = vkGetQueryPoolResults(
      device->getLogicalDevice(), queryPool, 0, query_count,
      queryResults.size() * sizeof(uint64_t), queryResults.data(),
      static_cast<VkDeviceSize>(sizeof(uint64_t)), VK_QUERY_RESULT_64_BIT);

  if (result != VK_NOT_READY) {
    pathTracingTiming = (static_cast<float>(queryResults[1] - queryResults[0]) *
                         timeStampPeriod) /
                        1000000.f;
  }
}

void PathTracing::cleanUp() {
  vkDestroyPipeline(device->getLogicalDevice(), pipeline, nullptr);
  vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);

  vkDestroyQueryPool(device->getLogicalDevice(), queryPool, nullptr);
}

PathTracing::~PathTracing() {}

void PathTracing::createQueryPool() {
  VkQueryPoolCreateInfo queryPoolInfo = {};
  queryPoolInfo.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
  // This query pool will store pipeline statistics
  queryPoolInfo.queryType = VK_QUERY_TYPE_TIMESTAMP;
  // Pipeline counters to be returned for this pool
  queryPoolInfo.pipelineStatistics =
      VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT;
  queryPoolInfo.queryCount = query_count;
  ASSERT_VULKAN(vkCreateQueryPool(device->getLogicalDevice(), &queryPoolInfo,
                                  NULL, &queryPool),
                "Failed to create query pool!");
}

void PathTracing::createPipeline(
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
  VkPushConstantRange push_constant_range{};
  push_constant_range.stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
  push_constant_range.offset = 0;
  push_constant_range.size = sizeof(PushConstantPathTracing);

  VkPipelineLayoutCreateInfo compute_pipeline_layout_create_info{};
  compute_pipeline_layout_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  compute_pipeline_layout_create_info.setLayoutCount =
      static_cast<uint32_t>(descriptorSetLayouts.size());
  compute_pipeline_layout_create_info.pushConstantRangeCount = 1;
  compute_pipeline_layout_create_info.pPushConstantRanges =
      &push_constant_range;
  compute_pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();

  ASSERT_VULKAN(vkCreatePipelineLayout(device->getLogicalDevice(),
                                       &compute_pipeline_layout_create_info,
                                       nullptr, &pipeline_layout),
                "Failed to create compute path tracing pipeline layout!");

  // create pipeline
  std::stringstream pathTracing_shader_dir;
  pathTracing_shader_dir « CMAKELISTS_DIR;
  pathTracing_shader_dir « "/Resources/Shader/path_tracing/";

  std::string pathTracing_shader = "path_tracing.comp";

  ShaderHelper shaderHelper;
  File pathTracingShaderFile(shaderHelper.getShaderSpvDir(
      pathTracing_shader_dir.str(), pathTracing_shader));
  std::vector<char> pathTracingShadercode =
      pathTracingShaderFile.readCharSequence();

  shaderHelper.compileShader(pathTracing_shader_dir.str(), pathTracing_shader);

  // build shader modules to link to graphics pipeline
  VkShaderModule pathTracingModule =
      shaderHelper.createShaderModule(device, pathTracingShadercode);

  // Specialization constant for workgroup size
  std::array<VkSpecializationMapEntry, 2> specEntries{};

  specEntries[0].constantID = 0;
  specEntries[0].size = sizeof(specializationData.specWorkGroupSizeX);
  specEntries[0].offset = 0;

  specEntries[1].constantID = 1;
  specEntries[1].size = sizeof(specializationData.specWorkGroupSizeY);
  specEntries[1].offset = offsetof(SpecializationData, specWorkGroupSizeY);

  // specEntries[2].constantID = 2;
  // specEntries[2].size = sizeof(specializationData.specWorkGroupSizeZ);
  // specEntries[2].offset = offsetof(SpecializationData, specWorkGroupSizeZ);

  VkSpecializationInfo specInfo{};
  specInfo.dataSize = sizeof(specializationData);
  specInfo.mapEntryCount = static_cast<uint32_t>(specEntries.size());
  specInfo.pMapEntries = specEntries.data();
  specInfo.pData = &specializationData;

  VkPipelineShaderStageCreateInfo compute_shader_integrate_create_info{};
  compute_shader_integrate_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  compute_shader_integrate_create_info.stage = VK_SHADER_STAGE_COMPUTE_BIT;
  compute_shader_integrate_create_info.module = pathTracingModule;
  compute_shader_integrate_create_info.pSpecializationInfo = &specInfo;
  compute_shader_integrate_create_info.pName = "main";

  // -- COMPUTE PIPELINE CREATION --
  VkComputePipelineCreateInfo compute_pipeline_create_info{};
  compute_pipeline_create_info.sType =
      VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
  compute_pipeline_create_info.stage = compute_shader_integrate_create_info;
  compute_pipeline_create_info.layout = pipeline_layout;
  compute_pipeline_create_info.flags = 0;
  // create compute pipeline
  ASSERT_VULKAN(vkCreateComputePipelines(
                    device->getLogicalDevice(), VK_NULL_HANDLE, 1,
                    &compute_pipeline_create_info, nullptr, &pipeline),
                "Failed to create a compute pipeline!");

  // Destroy shader modules, no longer needed after pipeline created
  vkDestroyShaderModule(device->getLogicalDevice(), pathTracingModule, nullptr);
}
```

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.

```cpp
#include "PostStage.h"

#include <array>
#include <vector>

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

PostStage::PostStage() {}

void PostStage::init(
    VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
  this->device = device;
  this->vulkanSwapChain = vulkanSwapChain;

  createOffscreenTextureSampler();

  createPushConstantRange();
  createDepthbufferImage();
  createRenderpass();
  createGraphicsPipeline(descriptorSetLayouts);
  createFramebuffer();
}

void PostStage::shaderHotReload(
    const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
  vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
  createGraphicsPipeline(descriptor_set_layouts);
}

void PostStage::recordCommands(
    VkCommandBuffer& commandBuffer, uint32_t image_index,
    const std::vector<VkDescriptorSet>& descriptorSets) {
  // information about how to begin a render pass (only needed for graphical
  // applications)
  VkRenderPassBeginInfo render_pass_begin_info{};
  render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
  render_pass_begin_info.renderPass = render_pass;  // render pass to begin
  render_pass_begin_info.renderArea.offset = {
      0, 0};  // start point of render pass in pixels
  const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
  render_pass_begin_info.renderArea.extent =
      swap_chain_extent;  // size of region to run render pass on (starting at
                          // offset)

  // make sure the order you put the values into the array matches with the
  // attchment order you have defined previous
  std::array<VkClearValue, 2> clear_values = {};
  clear_values[0].color = {0.2f, 0.65f, 0.4f, 1.0f};
  clear_values[1].depthStencil = {1.0f, 0};

  render_pass_begin_info.pClearValues = clear_values.data();
  render_pass_begin_info.clearValueCount =
      static_cast<uint32_t>(clear_values.size());

  // used framebuffer depends on the swap chain and therefore is changing for
  // each command buffer
  render_pass_begin_info.framebuffer = framebuffers[image_index];

  // begin render pass
  vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
                       VK_SUBPASS_CONTENTS_INLINE);
  auto aspectRatio = static_cast<float>(swap_chain_extent.width) /
                     static_cast<float>(swap_chain_extent.height);
  PushConstantPost pc_post{};
  pc_post.aspect_ratio = aspectRatio;
  vkCmdPushConstants(commandBuffer, pipeline_layout,
                     VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT,
                     0, sizeof(PushConstantPost), &pc_post);
  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
                    graphics_pipeline);
  vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
                          pipeline_layout, 0,
                          static_cast<uint32_t>(descriptorSets.size()),
                          descriptorSets.data(), 0, nullptr);
  vkCmdDraw(commandBuffer, 3, 1, 0, 0);

  // Rendering gui
  ImGui::Render();
  ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), commandBuffer);

  // end render pass
  vkCmdEndRenderPass(commandBuffer);
}

void PostStage::cleanUp() {
  depthBufferImage.cleanUp();
  for (auto framebuffer : framebuffers) {
    vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
  }

  vkDestroySampler(device->getLogicalDevice(), offscreenTextureSampler,
                   nullptr);

  vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
  vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
  vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
}

PostStage::~PostStage() {}

void PostStage::createDepthbufferImage() {
  // get supported format for depth buffer
  depth_format = choose_supported_format(
      device->getPhysicalDevice(),
      {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
       VK_FORMAT_D24_UNORM_S8_UINT},
      VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);

  // create depth buffer image
  // MIP LEVELS: for depth texture we only want 1 level :)
  const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
  depthBufferImage.createImage(device, swap_chain_extent.width,
                               swap_chain_extent.height, 1, depth_format,
                               VK_IMAGE_TILING_OPTIMAL,
                               VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
                               VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

  // depth buffer image view
  // MIP LEVELS: for depth texture we only want 1 level :)
  depthBufferImage.createImageView(device, depth_format,
                                   VK_IMAGE_ASPECT_DEPTH_BIT, 1);
}

void PostStage::createOffscreenTextureSampler() {
  // sampler create info
  VkSamplerCreateInfo sampler_create_info{};
  sampler_create_info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
  sampler_create_info.magFilter = VK_FILTER_LINEAR;
  sampler_create_info.minFilter = VK_FILTER_LINEAR;
  sampler_create_info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  sampler_create_info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  sampler_create_info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  sampler_create_info.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
  sampler_create_info.unnormalizedCoordinates = VK_FALSE;
  sampler_create_info.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
  sampler_create_info.mipLodBias = 0.0f;
  sampler_create_info.minLod = 0.0f;
  sampler_create_info.maxLod = 0.0f;
  sampler_create_info.anisotropyEnable = VK_TRUE;
  sampler_create_info.maxAnisotropy = 16;  // max anisotropy sample level

  VkResult result =
      vkCreateSampler(device->getLogicalDevice(), &sampler_create_info, nullptr,
                      &offscreenTextureSampler);
  ASSERT_VULKAN(result, "Failed to create a texture sampler!")
}

void PostStage::createPushConstantRange() {
  push_constant_range.stageFlags =
      VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
  push_constant_range.offset = 0;
  push_constant_range.size = sizeof(PushConstantPost);
}

void PostStage::createRenderpass() {
  // Color attachment of render pass
  VkAttachmentDescription color_attachment{};
  const VkFormat& swap_chain_image_format =
      vulkanSwapChain->getSwapChainFormat();
  color_attachment.format =
      swap_chain_image_format;  // format to use for attachment
  color_attachment.samples =
      VK_SAMPLE_COUNT_1_BIT;  // number of samples to write for multisampling
  color_attachment.loadOp =
      VK_ATTACHMENT_LOAD_OP_CLEAR;  // describes what to do with attachment
                                    // before rendering
  color_attachment.storeOp =
      VK_ATTACHMENT_STORE_OP_STORE;  // describes what to do with attachment
                                     // after rendering
```

```
color_attachment.stencilLoadOp =
    VK_ATTACHMENT_LOAD_OP_DONT_CARE;  // describes what to do with stencil
                                      // before rendering
color_attachment.stencilStoreOp =
    VK_ATTACHMENT_STORE_OP_DONT_CARE;  // describes what to do with stencil
                                       // after rendering

// framebuffer data will be stored as an image, but images can be given
// different layouts to give optimal use for certain operations
color_attachment.initialLayout =
    VK_IMAGE_LAYOUT_UNDEFINED;  // image data layout before render pass starts
color_attachment.finalLayout =
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;  // image data layout after render pass
                                      // (to change to)

// depth attachment of render pass
VkAttachmentDescription depth_attachment{};
depth_attachment.format = choose_supported_format(
    device->getPhysicalDevice(),
    {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
     VK_FORMAT_D24_UNORM_S8_UINT},
    VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);
depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
depth_attachment.finalLayout =
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

// attachment reference uses an attachment index that refers to index in the
// attachment list passed to renderPassCreateInfo
VkAttachmentReference color_attachment_reference{};
color_attachment_reference.attachment = 0;
color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

// attachment reference
VkAttachmentReference depth_attachment_reference{};
depth_attachment_reference.attachment = 1;
depth_attachment_reference.layout =
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

// information about a particular subpass the render pass is using
VkSubpassDescription subpass{};
subpass.pipelineBindPoint =
    VK_PIPELINE_BIND_POINT_GRAPHICS;  // pipeline type subpass is to be bound
                                      // to
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &color_attachment_reference;
subpass.pDepthStencilAttachment = &depth_attachment_reference;

// need to determine when layout transitions occur using subpass dependencies
std::array<VkSubpassDependency, 1> subpass_dependencies;

// conversion from VK_IMAGE_LAYOUT_UNDEFINED to
// VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
subpass_dependencies[0].srcSubpass =
    VK_SUBPASS_EXTERNAL;  // subpass index (VK_SUBPASS_EXTERNAL = Special
                          // value meaning outside of renderpass)
subpass_dependencies[0].srcStageMask =
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;  // pipeline stage
subpass_dependencies[0].srcAccessMask =
    VK_ACCESS_MEMORY_READ_BIT;  // stage access mask (memory access)
subpass_dependencies[0].dstSubpass = 0;
subpass_dependencies[0].dstStageMask =
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                                        VK_ACCESS_COLOR_ATTACHMENT_READ_BIT;
subpass_dependencies[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

std::array<VkAttachmentDescription, 2> render_pass_attachments = {
    color_attachment, depth_attachment};

// create info for render pass
VkRenderPassCreateInfo render_pass_create_info{};
render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
render_pass_create_info.attachmentCount =
    static_cast<uint32_t>(render_pass_attachments.size());
render_pass_create_info.pAttachments = render_pass_attachments.data();
render_pass_create_info.subpassCount = 1;
render_pass_create_info.pSubpasses = &subpass;
render_pass_create_info.dependencyCount =
    static_cast<uint32_t>(subpass_dependencies.size());
render_pass_create_info.pDependencies = subpass_dependencies.data();

VkResult result =
      vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
                         nullptr, &render_pass);
  ASSERT_VULKAN(result, "Failed to create render pass!")
}

void PostStage::createGraphicsPipeline(
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
  std::stringstream post_shader_dir;
  post_shader_dir « CMAKELISTS_DIR;
  post_shader_dir « "/Resources/Shader/post/";

  std::string post_vert_shader = "post.vert";
  std::string post_frag_shader = "post.frag";

  ShaderHelper shaderHelper;
  File vertexShaderFile(
      shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_vert_shader));
  std::vector<char> vertex_shader_code = vertexShaderFile.readCharSequence();
  File fragmentShaderFile(
      shaderHelper.getShaderSpvDir(post_shader_dir.str(), post_frag_shader));
  std::vector<char> fragment_shader_code =
      fragmentShaderFile.readCharSequence();

  shaderHelper.compileShader(post_shader_dir.str(), post_vert_shader);
  shaderHelper.compileShader(post_shader_dir.str(), post_frag_shader);

  // build shader modules to link to graphics pipeline
  VkShaderModule vertex_shader_module =
      shaderHelper.createShaderModule(device, vertex_shader_code);
  VkShaderModule fragment_shader_module =
      shaderHelper.createShaderModule(device, fragment_shader_code);

  // shader stage creation information
  // vertex stage creation information
  VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
  vertex_shader_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
  vertex_shader_create_info.module = vertex_shader_module;
  vertex_shader_create_info.pName = "main";

  // fragment stage creation information
  VkPipelineShaderStageCreateInfo fragment_shader_create_info{};
  fragment_shader_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
  fragment_shader_create_info.module = fragment_shader_module;
  fragment_shader_create_info.pName = "main";

  std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
      vertex_shader_create_info, fragment_shader_create_info};

  // how the data for a single vertex (including info such as position, color,
  // texture coords, normals, etc) is as a whole
  VkVertexInputBindingDescription binding_description{};
  binding_description.binding = 0;
  binding_description.stride = sizeof(Vertex);
  binding_description.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

  std::array<VkVertexInputAttributeDescription, 4> attribute_describtions =
      vertex::getVertexInputAttributeDesc();

  // CREATE PIPELINE
  // 1.) Vertex input
  VkPipelineVertexInputStateCreateInfo vertex_input_create_info{};
  vertex_input_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
  vertex_input_create_info.vertexBindingDescriptionCount = 0;
  vertex_input_create_info.pVertexBindingDescriptions = nullptr;
  vertex_input_create_info.vertexAttributeDescriptionCount = 0;
  vertex_input_create_info.pVertexAttributeDescriptions = nullptr;

  // input assembly
  VkPipelineInputAssemblyStateCreateInfo input_assembly{};
  input_assembly.sType =
      VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
  input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
  input_assembly.primitiveRestartEnable = VK_FALSE;

  // viewport & scissor
  // create a viewport info struct
  VkViewport viewport{};
  viewport.x = 0.0f;
  viewport.y = 0.0f;
  const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
  viewport.width = (float)swap_chain_extent.width;
  viewport.height = (float)swap_chain_extent.height;
  viewport.minDepth = 0.0f;
  viewport.maxDepth = 1.0f;

  // create a scissor info struct
  VkRect2D scissor{};
  scissor.offset = {0, 0};
  scissor.extent = swap_chain_extent;

  VkPipelineViewportStateCreateInfo viewport_state_create_info{};
  viewport_state_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
  viewport_state_create_info.viewportCount = 1;
  viewport_state_create_info.pViewports = &viewport;
  viewport_state_create_info.scissorCount = 1;
  viewport_state_create_info.pScissors = &scissor;

  // RASTERIZER
  VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
  rasterizer_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
  rasterizer_create_info.depthClampEnable = VK_FALSE;
  rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
  rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
  rasterizer_create_info.lineWidth = 1.0f;
  rasterizer_create_info.cullMode = VK_CULL_MODE_NONE;
  // winding to determine which side is front; y-coordinate is inverted in
  // comparison to OpenGL
  rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
  rasterizer_create_info.depthBiasClamp = VK_FALSE;

  // -- MULTISAMPLING --
  VkPipelineMultisampleStateCreateInfo multisample_create_info{};
  multisample_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
  multisample_create_info.sampleShadingEnable = VK_FALSE;
  multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;

  // -- BLENDING --
  // blend attachment state
  VkPipelineColorBlendAttachmentState color_state{};
  color_state.colorWriteMask =
      VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
      VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;

  color_state.blendEnable = VK_TRUE;
  // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
  // (dstColorBlendFactor * old_color)
  color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
  color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
  color_state.colorBlendOp = VK_BLEND_OP_ADD;
  color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
  color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
  color_state.alphaBlendOp = VK_BLEND_OP_ADD;

  VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
  color_blending_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
  color_blending_create_info.logicOpEnable =
      VK_FALSE;  // alternative to calculations is to use logical operations
  color_blending_create_info.logicOp = VK_LOGIC_OP_CLEAR;
  color_blending_create_info.attachmentCount = 1;
  color_blending_create_info.pAttachments = &color_state;
  for (int i = 0; i < 4; i++) {
    color_blending_create_info.blendConstants[0] = 0.f;
  }
  // -- PIPELINE LAYOUT --
  VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
  pipeline_layout_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  pipeline_layout_create_info.setLayoutCount =
      static_cast<uint32_t>(descriptorSetLayouts.size());
  pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
  pipeline_layout_create_info.pushConstantRangeCount = 1;
  pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;

  // create pipeline layout
  VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
                                           &pipeline_layout_create_info,
                                           nullptr, &pipeline_layout);
  ASSERT_VULKAN(result, "Failed to create pipeline layout!")

  // -- DEPTH STENCIL TESTING --
  VkPipelineDepthStencilStateCreateInfo depth_stencil_create_info{};
  depth_stencil_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
  depth_stencil_create_info.depthTestEnable = VK_TRUE;
  depth_stencil_create_info.depthWriteEnable = VK_TRUE;
```

```
  depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
  depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
  depth_stencil_create_info.stencilTestEnable = VK_FALSE;

  // -- GRAPHICS PIPELINE CREATION --
  VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
  graphics_pipeline_create_info.sType =
      VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
  graphics_pipeline_create_info.stageCount =
      static_cast<uint32_t>(shader_stages.size());
  graphics_pipeline_create_info.pStages = shader_stages.data();
  graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
  graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
  graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
  graphics_pipeline_create_info.pDynamicState = nullptr;
  graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
  graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
  graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
  graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
  graphics_pipeline_create_info.layout = pipeline_layout;
  graphics_pipeline_create_info.renderPass = render_pass;
  graphics_pipeline_create_info.subpass = 0;

  // pipeline derivatives : can create multiple pipelines that derive from one
  // another for optimization
  graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
  graphics_pipeline_create_info.basePipelineIndex = -1;

  // create graphics pipeline
  result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
                                     1, &graphics_pipeline_create_info, nullptr,
                                     &graphics_pipeline);
  ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")

  // Destroy shader modules, no longer needed after pipeline created
  vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
                        nullptr);
  vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,
                        nullptr);
}

void PostStage::createFramebuffer() {
  // resize framebuffer size to equal swap chain image count
  framebuffers.resize(vulkanSwapChain->getNumberSwapChainImages());

  for (size_t i = 0; i < vulkanSwapChain->getNumberSwapChainImages(); i++) {
    Texture& swap_chain_image = vulkanSwapChain->getSwapChainImage(i);

    std::array<VkImageView, 2> attachments = {swap_chain_image.getImageView(),
                                              depthBufferImage.getImageView()};

    VkFramebufferCreateInfo frame_buffer_create_info{};
    frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    frame_buffer_create_info.renderPass =
        render_pass;  // render pass layout the framebuffer will be used with
    frame_buffer_create_info.attachmentCount =
        static_cast<uint32_t>(attachments.size());
    frame_buffer_create_info.pAttachments =
        attachments.data();  // list of attachments (1:1 with render pass)
    const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
    frame_buffer_create_info.width =
        swap_chain_extent.width;  // framebuffer width
    frame_buffer_create_info.height =
        swap_chain_extent.height;         // framebuffer height
    frame_buffer_create_info.layers = 1;  // framebuffer layer

    VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
                                          &frame_buffer_create_info, nullptr,
                                          &framebuffers[i]);
    ASSERT_VULKAN(result, "Failed to create framebuffer!")
  }
}
```

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.

```
#include "Rasterizer.h"

#include <array>
#include <vector>

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

Rasterizer::Rasterizer() {}

void Rasterizer::init(
    VulkanDevice* device, VulkanSwapChain* vulkanSwapChain,
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts,
    VkCommandPool& commandPool) {
  this->device = device;
  this->vulkanSwapChain = vulkanSwapChain;

  createTextures(commandPool);
  createRenderPass();
  createPushConstantRange();
  createGraphicsPipeline(descriptorSetLayouts);
  createFramebuffer();
}

void Rasterizer::shaderHotReload(
    const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
  vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
  createGraphicsPipeline(descriptor_set_layouts);
}

Texture& Rasterizer::getOffscreenTexture(uint32_t index) {
  return offscreenTextures[index];
}

void Rasterizer::setPushConstant(PushConstantRasterizer pushConstant) {
  this->pushConstant = pushConstant;
}

void Rasterizer::recordCommands(
    VkCommandBuffer& commandBuffer, uint32_t image_index, Scene* scene,
    const std::vector<VkDescriptorSet>& descriptorSets) {
  // information about how to begin a render pass (only needed for graphical
  // applications)
  VkRenderPassBeginInfo render_pass_begin_info{};
  render_pass_begin_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
  render_pass_begin_info.renderPass = render_pass;
  render_pass_begin_info.renderArea.offset = {0, 0};
  const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
  render_pass_begin_info.renderArea.extent = swap_chain_extent;

  // make sure the order you put the values into the array matches with the
  // attchment order you have defined previous
  std::array<VkClearValue, 2> clear_values = {};
  clear_values[0].color = {0.2f, 0.65f, 0.4f, 1.0f};
  clear_values[1].depthStencil = {1.0f, 0};

  render_pass_begin_info.pClearValues = clear_values.data();
  render_pass_begin_info.clearValueCount =
      static_cast<uint32_t>(clear_values.size());
  render_pass_begin_info.framebuffer = framebuffer[image_index];

  // begin render pass
  vkCmdBeginRenderPass(commandBuffer, &render_pass_begin_info,
                       VK_SUBPASS_CONTENTS_INLINE);

  // bind pipeline to be used in render pass
  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
                    graphics_pipeline);

  for (uint32_t m = 0; m < static_cast<uint32_t>(scene->getModelCount()); m++) {
    // for GCC doen't allow references on rvalues go like that ...
    pushConstant.model = scene->getModelMatrix(0);
    // just "Push" constants to given shader stage directly (no buffer)
    vkCmdPushConstants(
        commandBuffer, pipeline_layout,
        VK_SHADER_STAGE_VERTEX_BIT,      // stage to push constants to
        0,                               // offset to push constants to update
        sizeof(PushConstantRasterizer),  // size of data being pushed
        &pushConstant);  // using model of current mesh (can be array)

    for (unsigned int k = 0; k < scene->getMeshCount(m); k++) {
      // list of vertex buffers we want to draw
      VkBuffer vertex_buffers[] = {
          scene->getVertexBuffer(m, k)};  // buffers to bind
      VkDeviceSize offsets[] = {0};
      vkCmdBindVertexBuffers(
          commandBuffer, 0, 1, vertex_buffers,
          offsets);  // command to bind vertex buffer before drawing with them

      // bind mesh index buffer with 0 offset and using the uint32 type
      vkCmdBindIndexBuffer(commandBuffer, scene->getIndexBuffer(m, k), 0,
                           VK_INDEX_TYPE_UINT32);

      // bind descriptor sets
      vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
                              pipeline_layout, 0,
                              static_cast<uint32_t>(descriptorSets.size()),
                              descriptorSets.data(), 0, nullptr);

      // execute pipeline
      vkCmdDrawIndexed(commandBuffer,
                       static_cast<uint32_t>(scene->getIndexCount(m, k)), 1, 0,
                       0, 0);
    }
  }

  // end render pass
  vkCmdEndRenderPass(commandBuffer);
}

void Rasterizer::cleanUp() {
  for (auto framebuffer : framebuffer) {
    vkDestroyFramebuffer(device->getLogicalDevice(), framebuffer, nullptr);
  }

  for (Texture texture : offscreenTextures) {
    texture.cleanUp();
  }

  depthBufferImage.cleanUp();

  vkDestroyPipeline(device->getLogicalDevice(), graphics_pipeline, nullptr);
  vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
  vkDestroyRenderPass(device->getLogicalDevice(), render_pass, nullptr);
}

Rasterizer::~Rasterizer() {}

void Rasterizer::createRenderPass() {
  // Color attachment of render pass
  VkAttachmentDescription color_attachment{};
  const VkFormat& swap_chain_image_format =
      vulkanSwapChain->getSwapChainFormat();
  color_attachment.format =
      swap_chain_image_format;  // format to use for attachment
  color_attachment.samples =
      VK_SAMPLE_COUNT_1_BIT;  // number of samples to write for multisampling
  color_attachment.loadOp =
      VK_ATTACHMENT_LOAD_OP_CLEAR;  // describes what to do with attachment
                                    // before rendering
  color_attachment.storeOp =
      VK_ATTACHMENT_STORE_OP_STORE;  // describes what to do with attachment
                                     // after rendering
  color_attachment.stencilLoadOp =
      VK_ATTACHMENT_LOAD_OP_DONT_CARE;  // describes what to do with stencil
                                        // before rendering
  color_attachment.stencilStoreOp =
      VK_ATTACHMENT_STORE_OP_DONT_CARE;  // describes what to do with stencil
                                         // after rendering

  // framebuffer data will be stored as an image, but images can be given
  // different layouts to give optimal use for certain operations
  color_attachment.initialLayout =
      VK_IMAGE_LAYOUT_GENERAL;  // image data layout before render pass starts
  color_attachment.finalLayout =
      VK_IMAGE_LAYOUT_GENERAL;  // image data layout after render pass (to
                                // change to)

  // depth attachment of render pass
  VkAttachmentDescription depth_attachment{};
  depth_attachment.format = choose_supported_format(
      device->getPhysicalDevice(),
      {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
       VK_FORMAT_D24_UNORM_S8_UINT},
      VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);

  depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;
  depth_attachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  depth_attachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  depth_attachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  depth_attachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  depth_attachment.finalLayout =
      VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

  // attachment reference uses an attachment index that refers to index in the
  // attachment list passed to renderPassCreateInfo
  VkAttachmentReference color_attachment_reference{};
  color_attachment_reference.attachment = 0;
  color_attachment_reference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

  // attachment reference
  VkAttachmentReference depth_attachment_reference{};
  depth_attachment_reference.attachment = 1;
  depth_attachment_reference.layout =
      VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

  // information about a particular subpass the render pass is using
  VkSubpassDescription subpass{};
  subpass.pipelineBindPoint =
      VK_PIPELINE_BIND_POINT_GRAPHICS;  // pipeline type subpass is to be bound
                                        // to
  subpass.colorAttachmentCount = 1;
  subpass.pColorAttachments = &color_attachment_reference;
  subpass.pDepthStencilAttachment = &depth_attachment_reference;

  // need to determine when layout transitions occur using subpass dependencies
  std::array<VkSubpassDependency, 1> subpass_dependencies;

  // conversion from VK_IMAGE_LAYOUT_UNDEFINED to
  // VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL transition must happen after ....
  subpass_dependencies[0].srcSubpass =
      VK_SUBPASS_EXTERNAL;  // subpass index (VK_SUBPASS_EXTERNAL = Special
                            // value meaning outside of renderpass)
  subpass_dependencies[0].srcStageMask =
      VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;  // pipeline stage
  subpass_dependencies[0].srcAccessMask =
      0;  // stage access mask (memory access)

  // but must happen before ...
  subpass_dependencies[0].dstSubpass = 0;
  subpass_dependencies[0].dstStageMask =
      VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
  subpass_dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
  subpass_dependencies[0].dependencyFlags = 0;  // VK_DEPENDENCY_BY_REGION_BIT;

  std::array<VkAttachmentDescription, 2> render_pass_attachments = {
      color_attachment, depth_attachment};

  // create info for render pass
  VkRenderPassCreateInfo render_pass_create_info{};
  render_pass_create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
  render_pass_create_info.attachmentCount =
      static_cast<uint32_t>(render_pass_attachments.size());
  render_pass_create_info.pAttachments = render_pass_attachments.data();
  render_pass_create_info.subpassCount = 1;
  render_pass_create_info.pSubpasses = &subpass;
  render_pass_create_info.dependencyCount =
      static_cast<uint32_t>(subpass_dependencies.size());
  render_pass_create_info.pDependencies = subpass_dependencies.data();

  VkResult result =
      vkCreateRenderPass(device->getLogicalDevice(), &render_pass_create_info,
                         nullptr, &render_pass);
  ASSERT_VULKAN(result, "Failed to create render pass!")
}

void Rasterizer::createFramebuffer() {
  framebuffer.resize(vulkanSwapChain->getNumberSwapChainImages());

  for (size_t i = 0; i < framebuffer.size(); i++) {
    std::array<VkImageView, 2> attachments = {
        offscreenTextures[i].getImageView(), depthBufferImage.getImageView()};

    VkFramebufferCreateInfo frame_buffer_create_info{};
    frame_buffer_create_info.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    frame_buffer_create_info.renderPass = render_pass;
    frame_buffer_create_info.attachmentCount =
        static_cast<uint32_t>(attachments.size());
    frame_buffer_create_info.pAttachments = attachments.data();
    const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
    frame_buffer_create_info.width = swap_chain_extent.width;
    frame_buffer_create_info.height = swap_chain_extent.height;
    frame_buffer_create_info.layers = 1;

    VkResult result = vkCreateFramebuffer(device->getLogicalDevice(),
                                          &frame_buffer_create_info, nullptr,
                                          &framebuffer[i]);
    ASSERT_VULKAN(result, "Failed to create framebuffer!");
  }
}

void Rasterizer::createPushConstantRange() {
  // define push constant values (no 'create' needed)
  push_constant_range.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
  push_constant_range.offset = 0;
  push_constant_range.size = sizeof(PushConstantRasterizer);
}

void Rasterizer::createTextures(VkCommandPool& commandPool) {
  offscreenTextures.resize(vulkanSwapChain->getNumberSwapChainImages());

  VkCommandBuffer cmdBuffer = commandBufferManager.beginCommandBuffer(
      device->getLogicalDevice(), commandPool);

  for (uint32_t index = 0;
       index <
       static_cast<uint32_t>(vulkanSwapChain->getNumberSwapChainImages());
       index++) {
    Texture texture{};
    const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
    const VkFormat& swap_chain_image_format =
        vulkanSwapChain->getSwapChainFormat();

    texture.createImage(
        device, swap_chain_extent.width, swap_chain_extent.height, 1,
        swap_chain_image_format, VK_IMAGE_TILING_OPTIMAL,
        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT |
            VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

    texture.createImageView(device, swap_chain_image_format,
                            VK_IMAGE_ASPECT_COLOR_BIT, 1);

    // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
    VulkanImage& image = texture.getVulkanImage();
    image.transitionImageLayout(cmdBuffer, VK_IMAGE_LAYOUT_UNDEFINED,
                                VK_IMAGE_LAYOUT_GENERAL, 1,
                                VK_IMAGE_ASPECT_COLOR_BIT);

    offscreenTextures[index] = texture;
  }

  VkFormat depth_format = choose_supported_format(
      device->getPhysicalDevice(),
      {VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
       VK_FORMAT_D24_UNORM_S8_UINT},
      VK_IMAGE_TILING_OPTIMAL, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT);

  // create depth buffer image
  // MIP LEVELS: for depth texture we only want 1 level :)
  const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
  depthBufferImage.createImage(device, swap_chain_extent.width,
                               swap_chain_extent.height, 1, depth_format,
                               VK_IMAGE_TILING_OPTIMAL,
                               VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
                               VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

  // depth buffer image view
  // MIP LEVELS: for depth texture we only want 1 level :)
  depthBufferImage.createImageView(
      device, depth_format,
      VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);

  // --- WE NEED A DIFFERENT LAYOUT FOR USAGE
  VulkanImage& vulkanImage = depthBufferImage.getVulkanImage();
  vulkanImage.transitionImageLayout(
      device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
      VK_IMAGE_LAYOUT_UNDEFINED,
      VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
      VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT, 1);

  commandBufferManager.endAndSubmitCommandBuffer(
      device->getLogicalDevice(), commandPool, device->getGraphicsQueue(),
      cmdBuffer);
}

void Rasterizer::createGraphicsPipeline(
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
  std::stringstream rasterizer_shader_dir;
  rasterizer_shader_dir « CMAKELISTS_DIR;
  rasterizer_shader_dir « "/Resources/Shader/rasterizer/";

  ShaderHelper shaderHelper;
  shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.vert");
  shaderHelper.compileShader(rasterizer_shader_dir.str(), "shader.frag");

  File vertexFile(
      shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.vert"));
  File fragmentFile(
      shaderHelper.getShaderSpvDir(rasterizer_shader_dir.str(), "shader.frag"));
  std::vector<char> vertex_shader_code = vertexFile.readCharSequence();
  std::vector<char> fragment_shader_code = fragmentFile.readCharSequence();

  // build shader modules to link to graphics pipeline
  VkShaderModule vertex_shader_module =
      shaderHelper.createShaderModule(device, vertex_shader_code);
  VkShaderModule fragment_shader_module =
      shaderHelper.createShaderModule(device, fragment_shader_code);

  // shader stage creation information
  // vertex stage creation information
  VkPipelineShaderStageCreateInfo vertex_shader_create_info{};
  vertex_shader_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  vertex_shader_create_info.stage = VK_SHADER_STAGE_VERTEX_BIT;
  vertex_shader_create_info.module = vertex_shader_module;
  vertex_shader_create_info.pName = "main";

  // fragment stage creation information
  VkPipelineShaderStageCreateInfo fragment_shader_create_info{};
  fragment_shader_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  fragment_shader_create_info.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
  fragment_shader_create_info.module = fragment_shader_module;
  fragment_shader_create_info.pName = "main";

  std::vector<VkPipelineShaderStageCreateInfo> shader_stages = {
      vertex_shader_create_info, fragment_shader_create_info};

  // how the data for a single vertex (including info such as position, color,
  // texture coords, normals, etc) is as a whole
  VkVertexInputBindingDescription binding_description{};
  binding_description.binding = 0;
  binding_description.stride = sizeof(Vertex);
  binding_description.inputRate =
      VK_VERTEX_INPUT_RATE_VERTEX;  // how to move between data after each
                                    // vertex.

  // how the data for an attribute is defined within a vertex
  std::array<VkVertexInputAttributeDescription, 4> attribute_describtions =
      vertex::getVertexInputAttributeDesc();

  // CREATE PIPELINE
  // 1.) Vertex input
  VkPipelineVertexInputStateCreateInfo vertex_input_create_info{};
  vertex_input_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
  vertex_input_create_info.vertexBindingDescriptionCount = 1;
  vertex_input_create_info.pVertexBindingDescriptions = &binding_description;
  vertex_input_create_info.vertexAttributeDescriptionCount =
      static_cast<uint32_t>(attribute_describtions.size());
  vertex_input_create_info.pVertexAttributeDescriptions =
      attribute_describtions.data();

  // input assembly
  VkPipelineInputAssemblyStateCreateInfo input_assembly{};
  input_assembly.sType =
      VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
  input_assembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
  input_assembly.primitiveRestartEnable = VK_FALSE;

  // viewport & scissor
  // create a viewport info struct
  VkViewport viewport{};
  viewport.x = 0.0f;
  viewport.y = 0.0f;
  const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
  viewport.width = (float)swap_chain_extent.width;
  viewport.height = (float)swap_chain_extent.height;
  viewport.minDepth = 0.0f;
  viewport.maxDepth = 1.0f;

  // create a scissor info struct
  VkRect2D scissor{};
  scissor.offset = {0, 0};
  scissor.extent = swap_chain_extent;

  VkPipelineViewportStateCreateInfo viewport_state_create_info{};
  viewport_state_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
  viewport_state_create_info.viewportCount = 1;
  viewport_state_create_info.pViewports = &viewport;
  viewport_state_create_info.scissorCount = 1;
  viewport_state_create_info.pScissors = &scissor;

  // RASTERIZER
  VkPipelineRasterizationStateCreateInfo rasterizer_create_info{};
  rasterizer_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
  rasterizer_create_info.depthClampEnable = VK_FALSE;
  rasterizer_create_info.rasterizerDiscardEnable = VK_FALSE;
  rasterizer_create_info.polygonMode = VK_POLYGON_MODE_FILL;
  rasterizer_create_info.lineWidth = 1.0f;
  rasterizer_create_info.cullMode = VK_CULL_MODE_BACK_BIT;  //
  // winding to determine which side is front; y-coordinate is inverted in
  // comparison to OpenGL
  rasterizer_create_info.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
  rasterizer_create_info.depthBiasClamp = VK_FALSE;

  // -- MULTISAMPLING --
  VkPipelineMultisampleStateCreateInfo multisample_create_info{};
  multisample_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
  multisample_create_info.sampleShadingEnable = VK_FALSE;
  multisample_create_info.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;

  // -- BLENDING --
  // blend attachment state
  VkPipelineColorBlendAttachmentState color_state{};
  color_state.colorWriteMask =
      VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
      VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;

  color_state.blendEnable = VK_TRUE;
  // blending uses equation: (srcColorBlendFactor * new_color) color_blend_op
  // (dstColorBlendFactor * old_color)
  color_state.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
  color_state.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
  color_state.colorBlendOp = VK_BLEND_OP_ADD;
  color_state.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
  color_state.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
  color_state.alphaBlendOp = VK_BLEND_OP_ADD;

  VkPipelineColorBlendStateCreateInfo color_blending_create_info{};
  color_blending_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
  color_blending_create_info.logicOpEnable = VK_FALSE;
  color_blending_create_info.attachmentCount = 1;
  color_blending_create_info.pAttachments = &color_state;

  // -- PIPELINE LAYOUT --
  VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
  pipeline_layout_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  pipeline_layout_create_info.setLayoutCount =
      static_cast<uint32_t>(descriptorSetLayouts.size());
  pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
  pipeline_layout_create_info.pushConstantRangeCount = 1;
  pipeline_layout_create_info.pPushConstantRanges = &push_constant_range;

  // create pipeline layout
  VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
                                           &pipeline_layout_create_info,
                                           nullptr, &pipeline_layout);
  ASSERT_VULKAN(result, "Failed to create pipeline layout!")

  // -- DEPTH STENCIL TESTING --
  VkPipelineDepthStencilStateCreateInfo depth_stencil_create_info{};
  depth_stencil_create_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
  depth_stencil_create_info.depthTestEnable = VK_TRUE;
  depth_stencil_create_info.depthWriteEnable = VK_TRUE;
```

```
  depth_stencil_create_info.depthCompareOp = VK_COMPARE_OP_LESS;
  depth_stencil_create_info.depthBoundsTestEnable = VK_FALSE;
  depth_stencil_create_info.stencilTestEnable = VK_FALSE;

  // -- GRAPHICS PIPELINE CREATION --
  VkGraphicsPipelineCreateInfo graphics_pipeline_create_info{};
  graphics_pipeline_create_info.sType =
      VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
  graphics_pipeline_create_info.stageCount =
      static_cast<uint32_t>(shader_stages.size());
  graphics_pipeline_create_info.pStages = shader_stages.data();
  graphics_pipeline_create_info.pVertexInputState = &vertex_input_create_info;
  graphics_pipeline_create_info.pInputAssemblyState = &input_assembly;
  graphics_pipeline_create_info.pViewportState = &viewport_state_create_info;
  graphics_pipeline_create_info.pDynamicState = nullptr;
  graphics_pipeline_create_info.pRasterizationState = &rasterizer_create_info;
  graphics_pipeline_create_info.pMultisampleState = &multisample_create_info;
  graphics_pipeline_create_info.pColorBlendState = &color_blending_create_info;
  graphics_pipeline_create_info.pDepthStencilState = &depth_stencil_create_info;
  graphics_pipeline_create_info.layout = pipeline_layout;
  graphics_pipeline_create_info.renderPass = render_pass;
  graphics_pipeline_create_info.subpass = 0;

  // pipeline derivatives : can create multiple pipelines that derive from one
  // another for optimization
  graphics_pipeline_create_info.basePipelineHandle = VK_NULL_HANDLE;
  graphics_pipeline_create_info.basePipelineIndex = -1;

  // create graphics pipeline
  result = vkCreateGraphicsPipelines(device->getLogicalDevice(), VK_NULL_HANDLE,
                                     1, &graphics_pipeline_create_info, nullptr,
                                     &graphics_pipeline);
  ASSERT_VULKAN(result, "Failed to create a graphics pipeline!")

  // Destroy shader modules, no longer needed after pipeline created
  vkDestroyShaderModule(device->getLogicalDevice(), vertex_shader_module,
                        nullptr);
  vkDestroyShaderModule(device->getLogicalDevice(), fragment_shader_module,
                        nullptr);
}
```

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.

```
#include "Raytracing.h"

#include <array>
#include <vector>

#include "File.h"
#include "MemoryHelper.h"
#include "ShaderHelper.h"

Raytracing::Raytracing() {}

void Raytracing::init(
    VulkanDevice* device,
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
  this->device = device;

  createPCRange();
  createGraphicsPipeline(descriptorSetLayouts);
  createSBT();
}

void Raytracing::shaderHotReload(
    const std::vector<VkDescriptorSetLayout>& descriptor_set_layouts) {
  vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
  createGraphicsPipeline(descriptor_set_layouts);
}

void Raytracing::recordCommands(
    VkCommandBuffer& commandBuffer, VulkanSwapChain* vulkanSwapChain,
    const std::vector<VkDescriptorSet>& descriptorSets) {
  uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
  uint32_t handle_size_aligned =
      align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);

  PFN_vkGetBufferDeviceAddressKHR vkGetBufferDeviceAddressKHR =
      reinterpret_cast<PFN_vkGetBufferDeviceAddressKHR>(vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkGetBufferDeviceAddressKHR"));

  PFN_vkCmdTraceRaysKHR pvkCmdTraceRaysKHR =
      (PFN_vkCmdTraceRaysKHR)vkGetDeviceProcAddr(device->getLogicalDevice(),
                                                 "vkCmdTraceRaysKHR");

  VkBufferDeviceAddressInfoKHR bufferDeviceAI{};
  bufferDeviceAI.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO;
  bufferDeviceAI.buffer = raygenShaderBindingTableBuffer.getBuffer();

  rgen_region.deviceAddress =
      vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
  rgen_region.stride = handle_size_aligned;
  rgen_region.size = handle_size_aligned;

  bufferDeviceAI.buffer = missShaderBindingTableBuffer.getBuffer();
  miss_region.deviceAddress =
      vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
  miss_region.stride = handle_size_aligned;
  miss_region.size = handle_size_aligned;

  bufferDeviceAI.buffer = hitShaderBindingTableBuffer.getBuffer();
  hit_region.deviceAddress =
      vkGetBufferDeviceAddressKHR(device->getLogicalDevice(), &bufferDeviceAI);
  hit_region.stride = handle_size_aligned;
  hit_region.size = handle_size_aligned;

  // for GCC doen't allow references on rvalues go like that ...
  pc.clear_color = {0.2f, 0.65f, 0.4f, 1.0f};
  // just "Push" constants to given shader stage directly (no buffer)
  vkCmdPushConstants(commandBuffer, pipeline_layout,
                     VK_SHADER_STAGE_RAYGEN_BIT_KHR |
                         VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
                         VK_SHADER_STAGE_MISS_BIT_KHR,
                     0, sizeof(PushConstantRaytracing), &pc);

  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
                    graphicsPipeline);

  vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
                          pipeline_layout, 0,
                          static_cast<uint32_t>(descriptorSets.size()),
                          descriptorSets.data(), 0, nullptr);

  const VkExtent2D& swap_chain_extent = vulkanSwapChain->getSwapChainExtent();
  pvkCmdTraceRaysKHR(commandBuffer, &rgen_region, &miss_region, &hit_region,
                     &call_region, swap_chain_extent.width,
                     swap_chain_extent.height, 1);
}

void Raytracing::cleanUp() {
  shaderBindingTableBuffer.cleanUp();
  raygenShaderBindingTableBuffer.cleanUp();
  missShaderBindingTableBuffer.cleanUp();
  hitShaderBindingTableBuffer.cleanUp();

  vkDestroyPipeline(device->getLogicalDevice(), graphicsPipeline, nullptr);
  vkDestroyPipelineLayout(device->getLogicalDevice(), pipeline_layout, nullptr);
}

Raytracing::~Raytracing() {}

void Raytracing::createPCRange() {
  // define push constant values (no 'create' needed)
  pc_ranges.stageFlags = VK_SHADER_STAGE_RAYGEN_BIT_KHR |
                         VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
                         VK_SHADER_STAGE_MISS_BIT_KHR;
  pc_ranges.offset = 0;
  pc_ranges.size = sizeof(PushConstantRaytracing);  // size of data being passed
}

void Raytracing::createGraphicsPipeline(
    const std::vector<VkDescriptorSetLayout>& descriptorSetLayouts) {
  PFN_vkCreateRayTracingPipelinesKHR pvkCreateRayTracingPipelinesKHR =
      (PFN_vkCreateRayTracingPipelinesKHR)vkGetDeviceProcAddr(
          device->getLogicalDevice(), "vkCreateRayTracingPipelinesKHR");

  std::stringstream raytracing_shader_dir;
  raytracing_shader_dir « CMAKELISTS_DIR;
  raytracing_shader_dir « "/Resources/Shader/raytracing/";

  std::string raygen_shader = "raytrace.rgen";
  std::string chit_shader = "raytrace.rchit";
  std::string miss_shader = "raytrace.rmiss";
  std::string shadow_shader = "shadow.rmiss";

  ShaderHelper shaderHelper;
  shaderHelper.compileShader(raytracing_shader_dir.str(), raygen_shader);
  shaderHelper.compileShader(raytracing_shader_dir.str(), chit_shader);
  shaderHelper.compileShader(raytracing_shader_dir.str(), miss_shader);
  shaderHelper.compileShader(raytracing_shader_dir.str(), shadow_shader);

  File raygenFile(
      shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), raygen_shader));
  File raychitFile(
      shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), chit_shader));
  File raymissFile(
      shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), miss_shader));
  File shadowFile(
      shaderHelper.getShaderSpvDir(raytracing_shader_dir.str(), shadow_shader));

  std::vector<char> raygen_shader_code = raygenFile.readCharSequence();
  std::vector<char> raychit_shader_code = raychitFile.readCharSequence();
  std::vector<char> raymiss_shader_code = raymissFile.readCharSequence();
  std::vector<char> shadow_shader_code = shadowFile.readCharSequence();

  // build shader modules to link to graphics pipeline
  VkShaderModule raygen_shader_module =
      shaderHelper.createShaderModule(device, raygen_shader_code);
  VkShaderModule raychit_shader_module =
      shaderHelper.createShaderModule(device, raychit_shader_code);
  VkShaderModule raymiss_shader_module =
      shaderHelper.createShaderModule(device, raymiss_shader_code);
  VkShaderModule shadow_shader_module =
      shaderHelper.createShaderModule(device, shadow_shader_code);

  // create all shader stage infos for creating a group
  VkPipelineShaderStageCreateInfo rgen_shader_stage_info{};
  rgen_shader_stage_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  rgen_shader_stage_info.stage = VK_SHADER_STAGE_RAYGEN_BIT_KHR;
  rgen_shader_stage_info.module = raygen_shader_module;
  rgen_shader_stage_info.pName = "main";

  VkPipelineShaderStageCreateInfo rmiss_shader_stage_info{};
  rmiss_shader_stage_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  rmiss_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
  rmiss_shader_stage_info.module = raymiss_shader_module;
  rmiss_shader_stage_info.pName = "main";

  VkPipelineShaderStageCreateInfo shadow_shader_stage_info{};
  shadow_shader_stage_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  shadow_shader_stage_info.stage = VK_SHADER_STAGE_MISS_BIT_KHR;
  shadow_shader_stage_info.module = shadow_shader_module;
  shadow_shader_stage_info.pName = "main";

  VkPipelineShaderStageCreateInfo rchit_shader_stage_info{};
  rchit_shader_stage_info.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  rchit_shader_stage_info.stage = VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR;
  rchit_shader_stage_info.module = raychit_shader_module;
  rchit_shader_stage_info.pName = "main";

  // we have all shader stages together
  std::array<VkPipelineShaderStageCreateInfo, 4> shader_stages = {
      rgen_shader_stage_info, rmiss_shader_stage_info, shadow_shader_stage_info,
      rchit_shader_stage_info};

  enum StageIndices { eRaygen, eMiss, eMiss2, eClosestHit, eShaderGroupCount };

  shader_groups.reserve(4);
  VkRayTracingShaderGroupCreateInfoKHR shader_group_create_infos[4];
```

```

shader_group_create_infos[0].sType =
    VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
shader_group_create_infos[0].pNext = nullptr;
shader_group_create_infos[0].type =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
shader_group_create_infos[0].generalShader = eRaygen;
shader_group_create_infos[0].closestHitShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[0].anyHitShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[0].intersectionShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[0].pShaderGroupCaptureReplayHandle = nullptr;

shader_groups.push_back(shader_group_create_infos[0]);

shader_group_create_infos[1].sType =
    VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
shader_group_create_infos[1].pNext = nullptr;
shader_group_create_infos[1].type =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
shader_group_create_infos[1].generalShader = eMiss;
shader_group_create_infos[1].closestHitShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[1].anyHitShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[1].intersectionShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[1].pShaderGroupCaptureReplayHandle = nullptr;

shader_groups.push_back(shader_group_create_infos[1]);

shader_group_create_infos[2].sType =
    VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
shader_group_create_infos[2].pNext = nullptr;
shader_group_create_infos[2].type =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
shader_group_create_infos[2].generalShader = eMiss2;
shader_group_create_infos[2].closestHitShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[2].anyHitShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[2].intersectionShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[2].pShaderGroupCaptureReplayHandle = nullptr;

shader_groups.push_back(shader_group_create_infos[2]);

shader_group_create_infos[3].sType =
    VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
shader_group_create_infos[3].pNext = nullptr;
shader_group_create_infos[3].type =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR;
shader_group_create_infos[3].generalShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[3].closestHitShader = eClosestHit;
shader_group_create_infos[3].anyHitShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[3].intersectionShader = VK_SHADER_UNUSED_KHR;
shader_group_create_infos[3].pShaderGroupCaptureReplayHandle = nullptr;

shader_groups.push_back(shader_group_create_infos[3]);

VkPipelineLayoutCreateInfo pipeline_layout_create_info{};
pipeline_layout_create_info.sType =
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipeline_layout_create_info.setLayoutCount =
    static_cast<uint32_t>(descriptorSetLayouts.size());
pipeline_layout_create_info.pSetLayouts = descriptorSetLayouts.data();
pipeline_layout_create_info.pushConstantRangeCount = 1;
pipeline_layout_create_info.pPushConstantRanges = &pc_ranges;

VkResult result = vkCreatePipelineLayout(device->getLogicalDevice(),
                                         &pipeline_layout_create_info,
                                         nullptr, &pipeline_layout);
ASSERT_VULKAN(result, "Failed to create raytracing pipeline layout!")

VkPipelineLibraryCreateInfoKHR pipeline_library_create_info{};
pipeline_library_create_info.sType =
    VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR;
pipeline_library_create_info.pNext = nullptr;
pipeline_library_create_info.libraryCount = 0;
pipeline_library_create_info.pLibraries = nullptr;

VkRayTracingPipelineCreateInfoKHR raytracing_pipeline_create_info{};
raytracing_pipeline_create_info.sType =
    VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR;
raytracing_pipeline_create_info.pNext = nullptr;
raytracing_pipeline_create_info.flags = 0;
raytracing_pipeline_create_info.stageCount =
    static_cast<uint32_t>(shader_stages.size());
raytracing_pipeline_create_info.pStages = shader_stages.data();
raytracing_pipeline_create_info.groupCount =
    static_cast<uint32_t>(shader_groups.size());
raytracing_pipeline_create_info.pGroups = shader_groups.data();
/*raytracing_pipeline_create_info.pLibraryInfo =
   &pipeline_library_create_info;
```

```
        raytracing_pipeline_create_info.pLibraryInterface = NULL;*/
  // TODO: HARDCODED FOR NOW;
  raytracing_pipeline_create_info.maxPipelineRayRecursionDepth = 2;
  raytracing_pipeline_create_info.layout = pipeline_layout;

  result = pvkCreateRayTracingPipelinesKHR(
      device->getLogicalDevice(), VK_NULL_HANDLE, VK_NULL_HANDLE, 1,
      &raytracing_pipeline_create_info, nullptr, &graphicsPipeline);

  ASSERT_VULKAN(result, "Failed to create raytracing pipeline!")

  vkDestroyShaderModule(device->getLogicalDevice(), raygen_shader_module,
                        nullptr);
  vkDestroyShaderModule(device->getLogicalDevice(), raymiss_shader_module,
                        nullptr);
  vkDestroyShaderModule(device->getLogicalDevice(), raychit_shader_module,
                        nullptr);
  vkDestroyShaderModule(device->getLogicalDevice(), shadow_shader_module,
                        nullptr);
}

void Raytracing::createSBT() {
  // load in functionality for raytracing shader group handles
  PFN_vkGetRayTracingShaderGroupHandlesKHR
      pvkGetRayTracingShaderGroupHandlesKHR =
          (PFN_vkGetRayTracingShaderGroupHandlesKHR)vkGetDeviceProcAddr(
              device->getLogicalDevice(),
              "vkGetRayTracingShaderGroupHandlesKHR");

  raytracing_properties = VkPhysicalDeviceRayTracingPipelinePropertiesKHR{};
  raytracing_properties.sType =
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR;

  VkPhysicalDeviceProperties2 properties{};
  properties.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;
  properties.pNext = &raytracing_properties;

  vkGetPhysicalDeviceProperties2(device->getPhysicalDevice(), &properties);

  uint32_t handle_size = raytracing_properties.shaderGroupHandleSize;
  uint32_t handle_size_aligned =
      align_up(handle_size, raytracing_properties.shaderGroupHandleAlignment);

  uint32_t group_count = static_cast<uint32_t>(shader_groups.size());
  uint32_t sbt_size = group_count * handle_size_aligned;

  std::vector<uint8_t> handles(sbt_size);

  VkResult result = pvkGetRayTracingShaderGroupHandlesKHR(
      device->getLogicalDevice(), graphicsPipeline, 0, group_count, sbt_size,
      handles.data());
  ASSERT_VULKAN(result, "Failed to get ray tracing shader group handles!")

  const VkBufferUsageFlags bufferUsageFlags =
      VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR |
      VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT;
  const VkMemoryPropertyFlags memoryUsageFlags =
      VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
      VK_MEMORY_PROPERTY_HOST_COHERENT_BIT;

  raygenShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
                                        memoryUsageFlags);

  missShaderBindingTableBuffer.create(device, 2 * handle_size, bufferUsageFlags,
                                      memoryUsageFlags);

  hitShaderBindingTableBuffer.create(device, handle_size, bufferUsageFlags,
                                     memoryUsageFlags);

  void* mapped_raygen = nullptr;
  vkMapMemory(device->getLogicalDevice(),
              raygenShaderBindingTableBuffer.getBufferMemory(), 0,
              VK_WHOLE_SIZE, 0, &mapped_raygen);

  void* mapped_miss = nullptr;
  vkMapMemory(device->getLogicalDevice(),
              missShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
              0, &mapped_miss);

  void* mapped_rchit = nullptr;
  vkMapMemory(device->getLogicalDevice(),
              hitShaderBindingTableBuffer.getBufferMemory(), 0, VK_WHOLE_SIZE,
              0, &mapped_rchit);

  memcpy(mapped_raygen, handles.data(), handle_size);
  memcpy(mapped_miss, handles.data() + handle_size_aligned, handle_size * 2);
  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.

```cpp
#include "VulkanRenderer.hpp"

#include <algorithm>
#include <iostream>
#include <vector>

#ifndef VMA_IMPLEMENTATION
#define VMA_IMPLEMENTATION
#endif  // !VMA_IMPLEMENTATION
#include <vk_mem_alloc.h>

#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>

#include <gsl/gsl>

#include "File.h"
#include "Globals.h"
#include "PushConstantPost.h"
#include "ShaderHelper.h"

VulkanRenderer::VulkanRenderer(Window* window, Scene* scene, GUI* gui,
                               Camera* camera)
    :

      window(window),
      scene(scene),
      gui(gui)

{
  updateUniforms(scene, camera, window);

  try {
    instance = VulkanInstance();

    VkDebugReportFlagsEXT debugReportFlags =
        VK_DEBUG_REPORT_ERROR_BIT_EXT | VK_DEBUG_REPORT_WARNING_BIT_EXT;
    if (ENABLE_VALIDATION_LAYERS)
      debug::setupDebugging(instance.getVulkanInstance(), debugReportFlags,
                            VK_NULL_HANDLE);

    create_surface();

    device = std::make_unique<VulkanDevice>(&instance, &surface);

    allocator =
        Allocator(device->getLogicalDevice(), device->getPhysicalDevice(),
                  instance.getVulkanInstance());

    create_command_pool();

    vulkanSwapChain.initVulkanContext(device.get(), window, surface);
    create_uniform_buffers();
    create_command_buffers();

    createSynchronization();

    createSharedRenderDescriptorSetLayouts();
    std::vector<VkDescriptorSetLayout> descriptor_set_layouts_rasterizer = {
        sharedRenderDescriptorSetLayout};
    rasterizer.init(device.get(), &vulkanSwapChain,
                    descriptor_set_layouts_rasterizer, graphics_command_pool);
    create_post_descriptor_layout();
    std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
        post_descriptor_set_layout};
    postStage.init(device.get(), &vulkanSwapChain, descriptor_set_layouts_post);
    createDescriptorPoolSharedRenderStages();
    createSharedRenderDescriptorSet();

    updatePostDescriptorSets();

    createRaytracingDescriptorPool();
    createRaytracingDescriptorSetLayouts();
    std::vector<VkDescriptorSetLayout> layouts;
    layouts.push_back(sharedRenderDescriptorSetLayout);
    layouts.push_back(raytracingDescriptorSetLayout);
    raytracingStage.init(device.get(), layouts);
    pathTracing.init(device.get(), layouts);

    scene->loadModel(device.get(), graphics_command_pool);
    updateTexturesInSharedRenderDescriptorSet();

    asManager.createASForScene(device.get(), graphics_command_pool, scene);
    create_object_description_buffer();
    createRaytracingDescriptorSets();
    updateRaytracingDescriptorSets();

    gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
                                 postStage.getRenderPass(),
                                 graphics_command_pool);

  } catch (const std::runtime_error& e) {
    printf("ERROR: %s\n", e.what());
  }
}

void VulkanRenderer::updateUniforms(Scene* scene, Camera* camera,
                                    Window* window) {
  const GUISceneSharedVars guiSceneSharedVars = scene->getGuiSceneSharedVars();

  globalUBO.view = camera->calculate_viewmatrix();
  globalUBO.projection =
      glm::perspective(glm::radians(camera->get_fov()),
                       (float)window->get_width() / (float)window->get_height(),
                       camera->get_near_plane(), camera->get_far_plane());

  sceneUBO.view_dir = glm::vec4(camera->get_camera_direction(), 1.0f);

  sceneUBO.light_dir =
      glm::vec4(guiSceneSharedVars.directional_light_direction[0],
                guiSceneSharedVars.directional_light_direction[1],
                guiSceneSharedVars.directional_light_direction[2], 1.0f);

  sceneUBO.cam_pos =
      glm::vec4(camera->get_camera_position(), camera->get_fov());
}

void VulkanRenderer::updateStateDueToUserInput(GUI* gui) {
  GUIRendererSharedVars& guiRendererSharedVars =
      gui->getGuiRendererSharedVars();

  if (guiRendererSharedVars.shader_hot_reload_triggered) {
    shaderHotReload();
    guiRendererSharedVars.shader_hot_reload_triggered = false;
  }
}

void VulkanRenderer::finishAllRenderCommands() {
  vkDeviceWaitIdle(device->getLogicalDevice());
}

void VulkanRenderer::shaderHotReload() {
  // wait until no actions being run on device before destroying
  vkDeviceWaitIdle(device->getLogicalDevice());

  std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
      sharedRenderDescriptorSetLayout};
  rasterizer.shaderHotReload(descriptor_set_layouts);

  std::vector<VkDescriptorSetLayout> descriptor_set_layouts_post = {
      post_descriptor_set_layout};
  postStage.shaderHotReload(descriptor_set_layouts_post);

  std::vector<VkDescriptorSetLayout> layouts = {sharedRenderDescriptorSetLayout,
                                                raytracingDescriptorSetLayout};
  raytracingStage.shaderHotReload(layouts);
  pathTracing.shaderHotReload(layouts);
}

void VulkanRenderer::drawFrame() {
  // We need to skip one frame
  // Due to ImGui need to call ImGui::NewFrame() again
  // if we recreated swapchain
  if (checkChangedFramebufferSize()) return;

  /*1. Get next available image to draw to and set something to signal when
     we're finished with the image  (a semaphore) wait for given fence to signal
     (open) from last draw before continuing*/
  VkResult result = vkWaitForFences(device->getLogicalDevice(), 1,
                                    &in_flight_fences[current_frame], VK_TRUE,
                                    std::numeric_limits<uint64_t>::max());
  ASSERT_VULKAN(result, "Failed to wait for fences!")
  // -- GET NEXT IMAGE --
  uint32_t image_index;
  result = vkAcquireNextImageKHR(
      device->getLogicalDevice(), vulkanSwapChain.getSwapChain(),
      std::numeric_limits<uint64_t>::max(), image_available[current_frame],
      VK_NULL_HANDLE, &image_index);

  if (result == VK_ERROR_OUT_OF_DATE_KHR) {
    // recreate_swap_chain();
    return;

  } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
    throw std::runtime_error("Failed to acquire next image!");
  }

  //// check if previous frame is using this image (i.e. there is its fence to
  /// wait on)
  if (images_in_flight_fences[image_index] != VK_NULL_HANDLE) {
    vkWaitForFences(device->getLogicalDevice(), 1,
                    &images_in_flight_fences[image_index], VK_TRUE, UINT64_MAX);
  }

  // mark the image as now being in use by this frame
  images_in_flight_fences[image_index] = in_flight_fences[current_frame];

  VkCommandBufferBeginInfo buffer_begin_info{};
  buffer_begin_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
  buffer_begin_info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
  // start recording commands to command buffer
  result =
      vkBeginCommandBuffer(command_buffers[image_index], &buffer_begin_info);
  ASSERT_VULKAN(result, "Failed to start recording a command buffer!")

  update_uniform_buffers(image_index);

  GUIRendererSharedVars& guiRendererSharedVars =
      gui->getGuiRendererSharedVars();
  if (guiRendererSharedVars.raytracing)
    update_raytracing_descriptor_set(image_index);

  record_commands(image_index);

  // stop recording to command buffer
  result = vkEndCommandBuffer(command_buffers[image_index]);
  ASSERT_VULKAN(result, "Failed to stop recording a command buffer!")

  // 2. Submit command buffer to queue for execution, making sure it waits for
  // the image to be signalled as available before drawing and signals when it
  // has finished rendering
  // -- SUBMIT COMMAND BUFFER TO RENDER --
  VkSubmitInfo submit_info{};
  submit_info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  submit_info.waitSemaphoreCount = 1;  // number of semaphores to wait on
  submit_info.pWaitSemaphores =
      &image_available[current_frame];  // list of semaphores to wait on

  VkPipelineStageFlags wait_stages = {

      VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT /*|
                  VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT |
                  VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR*/

  };

  submit_info.pWaitDstStageMask =
      &wait_stages;  // stages to check semaphores at

  submit_info.commandBufferCount = 1;  // number of command buffers to submit
  submit_info.pCommandBuffers =
      &command_buffers[image_index];     // command buffer to submit
  submit_info.signalSemaphoreCount = 1;  // number of semaphores to signal
  submit_info.pSignalSemaphores =
      &render_finished[current_frame];  // semaphores to signal when command
                                        // buffer finishes

  result = vkResetFences(device->getLogicalDevice(), 1,
                         &in_flight_fences[current_frame]);
  ASSERT_VULKAN(result, "Failed to reset fences!")

  // submit command buffer to queue
  result = vkQueueSubmit(device->getGraphicsQueue(), 1, &submit_info,
                         in_flight_fences[current_frame]);
  ASSERT_VULKAN(result, "Failed to submit command buffer to queue!")

  // 3. Present image to screen when it has signalled finished rendering
  // -- PRESENT RENDERED IMAGE TO SCREEN --
  VkPresentInfoKHR present_info{};
  present_info.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
  present_info.waitSemaphoreCount = 1;  // number of semaphores to wait on
  present_info.pWaitSemaphores =
      &render_finished[current_frame];  // semaphores to wait on
  present_info.swapchainCount = 1;      // number of swapchains to present to
  const VkSwapchainKHR swapchain = vulkanSwapChain.getSwapChain();
  present_info.pSwapchains = &swapchain;  // swapchains to present images to
  present_info.pImageIndices =
      &image_index;  // index of images in swapchain to present

  result = vkQueuePresentKHR(device->getPresentationQueue(), &present_info);

  if (result == VK_ERROR_OUT_OF_DATE_KHR) {
    // recreate_swap_chain();
    return;

  } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
    throw std::runtime_error("Failed to acquire next image!");
  }

  if (result != VK_SUCCESS) {
    throw std::runtime_error("Failed to submit to present queue!");
  }

  current_frame = (current_frame + 1) % MAX_FRAME_DRAWS;
}

void VulkanRenderer::create_surface() {
  // create surface (creates a surface create info struct, runs the create
  // surface function, returns result)
  ASSERT_VULKAN(
      glfwCreateWindowSurface(instance.getVulkanInstance(),
                              window->get_window(), nullptr, &surface),
      "Failed to create a surface!");
}

void VulkanRenderer::create_post_descriptor_layout() {
  // UNIFORM VALUES DESCRIPTOR SET LAYOUT
  // globalUBO Binding info
  VkDescriptorSetLayoutBinding post_sampler_layout_binding{};
  post_sampler_layout_binding.binding =
      0;  // binding point in shader (designated by binding number in shader)
  post_sampler_layout_binding.descriptorType =
      VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;  // type of descriptor
                                                  // (uniform, dynamic uniform,
                                                  // image sampler, etc)
  post_sampler_layout_binding.descriptorCount =
      1;  // number of descriptors for binding
  post_sampler_layout_binding.stageFlags =
      VK_SHADER_STAGE_FRAGMENT_BIT;  // we need to say at which shader we bind
                                     // this uniform to
  post_sampler_layout_binding.pImmutableSamplers =
      nullptr;  // for texture: can make sampler data unchangeable (immutable)
                // by specifying in layout

  std::vector<VkDescriptorSetLayoutBinding> layout_bindings = {
      post_sampler_layout_binding};

  // create descriptor set layout with given bindings
  VkDescriptorSetLayoutCreateInfo layout_create_info{};
  layout_create_info.sType =
      VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
  layout_create_info.bindingCount = static_cast<uint32_t>(
      layout_bindings.size());  // only have 1 for the globalUBO
  layout_create_info.pBindings =
      layout_bindings.data();  // array of binding infos

  // create descriptor set layout
  VkResult result = vkCreateDescriptorSetLayout(device->getLogicalDevice(),
                                                &layout_create_info, nullptr,
                                                &post_descriptor_set_layout);
  ASSERT_VULKAN(result, "Failed to create descriptor set layout!")

  VkDescriptorPoolSize post_pool_size{};
  post_pool_size.type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
  post_pool_size.descriptorCount = static_cast<uint32_t>(1);

  // list of pool sizes
  std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {post_pool_size};

  VkDescriptorPoolCreateInfo pool_create_info{};
  pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
  pool_create_info.maxSets =
      vulkanSwapChain
          .getNumberSwapChainImages();  // maximum number of descriptor sets
                                        // that can be created from pool
  pool_create_info.poolSizeCount = static_cast<uint32_t>(
      descriptor_pool_sizes.size());  // amount of pool sizes being passed
  pool_create_info.pPoolSizes =
      descriptor_pool_sizes.data();  // pool sizes to create pool with

  // create descriptor pool
  result = vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
                                  nullptr, &post_descriptor_pool);
  ASSERT_VULKAN(result, "Failed to create a descriptor pool!")

  // resize descriptor set list so one for every buffer
  post_descriptor_set.resize(vulkanSwapChain.getNumberSwapChainImages());

  std::vector<VkDescriptorSetLayout> set_layouts(
      vulkanSwapChain.getNumberSwapChainImages(), post_descriptor_set_layout);

  // descriptor set allocation info
  VkDescriptorSetAllocateInfo set_alloc_info{};
  set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
  set_alloc_info.descriptorPool =
      post_descriptor_pool;  // pool to allocate descriptor set from
  set_alloc_info.descriptorSetCount =
      vulkanSwapChain.getNumberSwapChainImages();  // number of sets to allocate
  set_alloc_info.pSetLayouts =
      set_layouts.data();  // layouts to use to allocate sets (1:1 relationship)

  // allocate descriptor sets (multiple)
  result = vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
                                    post_descriptor_set.data());
  ASSERT_VULKAN(result, "Failed to create descriptor sets!")
}

void VulkanRenderer::updatePostDescriptorSets() {
  // update all of descriptor set buffer bindings
  for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
    // texture image info
    VkDescriptorImageInfo image_info{};
    image_info.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    Texture& renderResult = rasterizer.getOffscreenTexture(i);
    image_info.imageView = renderResult.getImageView();
    image_info.sampler = postStage.getOffscreenSampler();

    // descriptor write info
    VkWriteDescriptorSet descriptor_write{};
    descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    descriptor_write.dstSet = post_descriptor_set[i];
    descriptor_write.dstBinding = 0;
    descriptor_write.dstArrayElement = 0;
    descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    descriptor_write.descriptorCount = 1;
    descriptor_write.pImageInfo = &image_info;

    // update new descriptor set
    vkUpdateDescriptorSets(device->getLogicalDevice(), 1, &descriptor_write, 0,
                           nullptr);
  }
}

void VulkanRenderer::createRaytracingDescriptorPool() {
  std::array<VkDescriptorPoolSize, 2> descriptor_pool_sizes{};

  descriptor_pool_sizes[0].type = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
  descriptor_pool_sizes[0].descriptorCount = 1;

  descriptor_pool_sizes[1].type = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
  descriptor_pool_sizes[1].descriptorCount = 1;

  VkDescriptorPoolCreateInfo descriptor_pool_create_info{};
  descriptor_pool_create_info.sType =
      VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
  descriptor_pool_create_info.poolSizeCount =
      static_cast<uint32_t>(descriptor_pool_sizes.size());
  descriptor_pool_create_info.pPoolSizes = descriptor_pool_sizes.data();
  descriptor_pool_create_info.maxSets =
      vulkanSwapChain.getNumberSwapChainImages();

  VkResult result = vkCreateDescriptorPool(device->getLogicalDevice(),
                                           &descriptor_pool_create_info,
                                           nullptr, &raytracingDescriptorPool);
  ASSERT_VULKAN(result, "Failed to create command pool!")
}

void VulkanRenderer::cleanUpSync() {
  for (int i = 0; i < MAX_FRAME_DRAWS; i++) {
    vkDestroySemaphore(device->getLogicalDevice(), render_finished[i], nullptr);
    vkDestroySemaphore(device->getLogicalDevice(), image_available[i], nullptr);
    vkDestroyFence(device->getLogicalDevice(), in_flight_fences[i], nullptr);
  }
}

void VulkanRenderer::create_object_description_buffer() {
  std::vector<ObjectDescription> objectDescriptions =
      scene->getObjectDescriptions();

  vulkanBufferManager.createBufferAndUploadVectorOnDevice(
      device.get(), graphics_command_pool, objectDescriptionBuffer,
      VK_BUFFER_USAGE_TRANSFER_DST_BIT |
          VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
          VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
      VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
          VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
      objectDescriptions);

  // update the object description set
  // update all of descriptor set buffer bindings
  for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
    VkDescriptorBufferInfo object_descriptions_buffer_info{};
    // image_info.sampler = VK_DESCRIPTOR_TYPE_SAMPLER;
    object_descriptions_buffer_info.buffer =
        objectDescriptionBuffer.getBuffer();
    object_descriptions_buffer_info.offset = 0;
    object_descriptions_buffer_info.range = VK_WHOLE_SIZE;

    VkWriteDescriptorSet descriptor_object_descriptions_writer{};
    descriptor_object_descriptions_writer.sType =
        VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    descriptor_object_descriptions_writer.pNext = nullptr;
    descriptor_object_descriptions_writer.dstSet = sharedRenderDescriptorSet[i];
    descriptor_object_descriptions_writer.dstBinding =
        OBJECT_DESCRIPTION_BINDING;
    descriptor_object_descriptions_writer.dstArrayElement = 0;
    descriptor_object_descriptions_writer.descriptorCount = 1;
    descriptor_object_descriptions_writer.descriptorType =
        VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
    descriptor_object_descriptions_writer.pImageInfo = nullptr;
    descriptor_object_descriptions_writer.pBufferInfo =
        &object_descriptions_buffer_info;
    descriptor_object_descriptions_writer.pTexelBufferView =
        nullptr;  // information about buffer data to bind

    std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
        descriptor_object_descriptions_writer};

    // update the descriptor sets with new buffer/binding info
    vkUpdateDescriptorSets(device->getLogicalDevice(),
                           static_cast<uint32_t>(write_descriptor_sets.size()),
                           write_descriptor_sets.data(), 0, nullptr);
  }
}

void VulkanRenderer::createRaytracingDescriptorSetLayouts() {
  {
    std::array<VkDescriptorSetLayoutBinding, 2>
        descriptor_set_layout_bindings{};

    // here comes the top level acceleration structure
    descriptor_set_layout_bindings[0].binding = TLAS_BINDING;
    descriptor_set_layout_bindings[0].descriptorCount = 1;
    descriptor_set_layout_bindings[0].descriptorType =
        VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
    descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;
    // load them into the raygeneration and chlosest hit shader
    descriptor_set_layout_bindings[0].stageFlags =
        VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
        VK_SHADER_STAGE_COMPUTE_BIT;
    // here comes to previous rendered image
    descriptor_set_layout_bindings[1].binding = OUT_IMAGE_BINDING;
    descriptor_set_layout_bindings[1].descriptorCount = 1;
    descriptor_set_layout_bindings[1].descriptorType =
        VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
    descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;
    // load them into the raygeneration and chlosest hit shader
    descriptor_set_layout_bindings[1].stageFlags =
        VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
        VK_SHADER_STAGE_COMPUTE_BIT;

    VkDescriptorSetLayoutCreateInfo descriptor_set_layout_create_info{};
    descriptor_set_layout_create_info.sType =
        VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    descriptor_set_layout_create_info.bindingCount =
        static_cast<uint32_t>(descriptor_set_layout_bindings.size());
    descriptor_set_layout_create_info.pBindings =
        descriptor_set_layout_bindings.data();

    VkResult result = vkCreateDescriptorSetLayout(
        device->getLogicalDevice(), &descriptor_set_layout_create_info, nullptr,
        &raytracingDescriptorSetLayout);
    ASSERT_VULKAN(result, "Failed to create raytracing descriptor set layout!")
  }
}

void VulkanRenderer::createRaytracingDescriptorSets() {
  // resize descriptor set list so one for every buffer
  raytracingDescriptorSet.resize(vulkanSwapChain.getNumberSwapChainImages());

  std::vector<VkDescriptorSetLayout> set_layouts(
      vulkanSwapChain.getNumberSwapChainImages(),
      raytracingDescriptorSetLayout);

  VkDescriptorSetAllocateInfo descriptor_set_allocate_info{};
  descriptor_set_allocate_info.sType =
      VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
  ;
  descriptor_set_allocate_info.descriptorPool = raytracingDescriptorPool;
  descriptor_set_allocate_info.descriptorSetCount =
      vulkanSwapChain.getNumberSwapChainImages();
  descriptor_set_allocate_info.pSetLayouts = set_layouts.data();

  VkResult result = vkAllocateDescriptorSets(device->getLogicalDevice(),
                                             &descriptor_set_allocate_info,
                                             raytracingDescriptorSet.data());
  ASSERT_VULKAN(result, "Failed to allocate raytracing descriptor set!")
}

void VulkanRenderer::updateRaytracingDescriptorSets() {
  for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
    VkWriteDescriptorSetAccelerationStructureKHR
        descriptor_set_acceleration_structure{};
    descriptor_set_acceleration_structure.sType =
        VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
    descriptor_set_acceleration_structure.pNext = nullptr;
    descriptor_set_acceleration_structure.accelerationStructureCount = 1;
    VkAccelerationStructureKHR& vulkanTLAS = asManager.getTLAS();
    descriptor_set_acceleration_structure.pAccelerationStructures = &vulkanTLAS;

    VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
    write_descriptor_set_acceleration_structure.sType =
        VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    write_descriptor_set_acceleration_structure.pNext =
        &descriptor_set_acceleration_structure;
    write_descriptor_set_acceleration_structure.dstSet =
        raytracingDescriptorSet[i];
    write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
    write_descriptor_set_acceleration_structure.dstArrayElement = 0;
    write_descriptor_set_acceleration_structure.descriptorCount = 1;
    write_descriptor_set_acceleration_structure.descriptorType =
        VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
    write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
    write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
    write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;

    VkDescriptorImageInfo image_info{};
    Texture& renderResult = rasterizer.getOffscreenTexture(i);
    image_info.imageView = renderResult.getImageView();
    image_info.imageLayout = VK_IMAGE_LAYOUT_GENERAL;

    VkWriteDescriptorSet descriptor_image_writer{};
    descriptor_image_writer.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    descriptor_image_writer.pNext = nullptr;
    descriptor_image_writer.dstSet = raytracingDescriptorSet[i];
    descriptor_image_writer.dstBinding = OUT_IMAGE_BINDING;
    descriptor_image_writer.dstArrayElement = 0;
    descriptor_image_writer.descriptorCount = 1;
    descriptor_image_writer.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
    descriptor_image_writer.pImageInfo = &image_info;
    descriptor_image_writer.pBufferInfo = nullptr;
    descriptor_image_writer.pTexelBufferView = nullptr;

    std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
        write_descriptor_set_acceleration_structure, descriptor_image_writer};

    // update the descriptor sets with new buffer/binding info
    vkUpdateDescriptorSets(device->getLogicalDevice(),
                           static_cast<uint32_t>(write_descriptor_sets.size()),
                           write_descriptor_sets.data(), 0, nullptr);
  }
}

void VulkanRenderer::createSharedRenderDescriptorSetLayouts() {
  std::array<VkDescriptorSetLayoutBinding, 5> descriptor_set_layout_bindings{};
  // UNIFORM VALUES DESCRIPTOR SET LAYOUT
  // globalUBO Binding info
  descriptor_set_layout_bindings[0].binding = globalUBO_BINDING;
  descriptor_set_layout_bindings[0].descriptorType =
      VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
  descriptor_set_layout_bindings[0].descriptorCount = 1;
  descriptor_set_layout_bindings[0].stageFlags =
      VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_RAYGEN_BIT_KHR |
      VK_SHADER_STAGE_COMPUTE_BIT;
  descriptor_set_layout_bindings[0].pImmutableSamplers = nullptr;

  // our model matrix which updates every frame for each object
  descriptor_set_layout_bindings[1].binding = sceneUBO_BINDING;
  descriptor_set_layout_bindings[1].descriptorType =
      VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
  descriptor_set_layout_bindings[1].descriptorCount = 1;
  descriptor_set_layout_bindings[1].stageFlags =
      VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
      VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
      VK_SHADER_STAGE_COMPUTE_BIT;
  descriptor_set_layout_bindings[1].pImmutableSamplers = nullptr;

  descriptor_set_layout_bindings[2].binding = OBJECT_DESCRIPTION_BINDING;
  descriptor_set_layout_bindings[2].descriptorCount = 1;
  descriptor_set_layout_bindings[2].descriptorType =
      VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
  descriptor_set_layout_bindings[2].pImmutableSamplers = nullptr;
  // load them into the raygeneration and chlosest hit shader
  descriptor_set_layout_bindings[2].stageFlags =
      VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT |
      VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR | VK_SHADER_STAGE_COMPUTE_BIT;

  // CREATE TEXTURE SAMPLER DESCRIPTOR SET LAYOUT
  // texture binding info
  descriptor_set_layout_bindings[3].binding = SAMPLER_BINDING;
  descriptor_set_layout_bindings[3].descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
  descriptor_set_layout_bindings[3].descriptorCount = MAX_TEXTURE_COUNT;
  descriptor_set_layout_bindings[3].stageFlags =
      VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
      VK_SHADER_STAGE_COMPUTE_BIT;
  descriptor_set_layout_bindings[3].pImmutableSamplers = nullptr;

  descriptor_set_layout_bindings[4].binding = TEXTURES_BINDING;
  descriptor_set_layout_bindings[4].descriptorType =
      VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
  descriptor_set_layout_bindings[4].descriptorCount = MAX_TEXTURE_COUNT;
  descriptor_set_layout_bindings[4].stageFlags =
      VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR |
      VK_SHADER_STAGE_COMPUTE_BIT;
  descriptor_set_layout_bindings[4].pImmutableSamplers = nullptr;

  // create descriptor set layout with given bindings
  VkDescriptorSetLayoutCreateInfo layout_create_info{};
  layout_create_info.sType =
      VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
  layout_create_info.bindingCount =
      static_cast<uint32_t>(descriptor_set_layout_bindings.size());
  layout_create_info.pBindings = descriptor_set_layout_bindings.data();

  // create descriptor set layout
  VkResult result = vkCreateDescriptorSetLayout(
      device->getLogicalDevice(), &layout_create_info, nullptr,
      &sharedRenderDescriptorSetLayout);
  ASSERT_VULKAN(result, "Failed to create descriptor set layout!")
}

void VulkanRenderer::create_command_pool() {
  // get indices of queue familes from device
  QueueFamilyIndices queue_family_indices = device->getQueueFamilies();

  {
    VkCommandPoolCreateInfo pool_info{};
    pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    pool_info.flags =
        VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;  // we are ready now to
                                                          // re-record our
                                                          // command buffers
    pool_info.queueFamilyIndex =
        queue_family_indices
            .graphics_family;  // queue family type that buffers from this
                               // command pool will use

    // create a graphics queue family command pool
    VkResult result =
        vkCreateCommandPool(device->getLogicalDevice(), &pool_info, nullptr,
                            &graphics_command_pool);
    ASSERT_VULKAN(result, "Failed to create command pool!")
  }

  {
    VkCommandPoolCreateInfo pool_info{};
    pool_info.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    pool_info.flags =
        VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;  // we are ready now to
                                                          // re-record our
                                                          // command buffers
    pool_info.queueFamilyIndex =
        queue_family_indices.compute_family;  // queue family type that buffers
                                              // from this command pool will use

    // create a graphics queue family command pool
    VkResult result = vkCreateCommandPool(
        device->getLogicalDevice(), &pool_info, nullptr, &compute_command_pool);
    ASSERT_VULKAN(result, "Failed to create command pool!")
  }
}

void VulkanRenderer::cleanUpCommandPools() {
  vkDestroyCommandPool(device->getLogicalDevice(), graphics_command_pool,
                       nullptr);
  vkDestroyCommandPool(device->getLogicalDevice(), compute_command_pool,
                       nullptr);
}

void VulkanRenderer::create_command_buffers() {
  // resize command buffer count to have one for each framebuffer
  command_buffers.resize(vulkanSwapChain.getNumberSwapChainImages());

  VkCommandBufferAllocateInfo command_buffer_alloc_info{};
  command_buffer_alloc_info.sType =
      VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
  command_buffer_alloc_info.commandPool = graphics_command_pool;
  command_buffer_alloc_info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;

  command_buffer_alloc_info.commandBufferCount =
      static_cast<uint32_t>(command_buffers.size());

  VkResult result = vkAllocateCommandBuffers(device->getLogicalDevice(),
                                             &command_buffer_alloc_info,
                                             command_buffers.data());
  ASSERT_VULKAN(result, "Failed to allocate command buffers!")
}

void VulkanRenderer::createSynchronization() {
  image_available.resize(vulkanSwapChain.getNumberSwapChainImages(),
                         VK_NULL_HANDLE);
  render_finished.resize(vulkanSwapChain.getNumberSwapChainImages(),
                         VK_NULL_HANDLE);
  in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
                          VK_NULL_HANDLE);
  images_in_flight_fences.resize(vulkanSwapChain.getNumberSwapChainImages(),
                                 VK_NULL_HANDLE);

  // semaphore creation information
  VkSemaphoreCreateInfo semaphore_create_info{};
  semaphore_create_info.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

  // fence creation information
  VkFenceCreateInfo fence_create_info{};
  fence_create_info.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
  fence_create_info.flags = VK_FENCE_CREATE_SIGNALED_BIT;

  for (int i = 0; i < MAX_FRAME_DRAWS; i++) {
    if ((vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
                           nullptr, &image_available[i]) != VK_SUCCESS) ||
        (vkCreateSemaphore(device->getLogicalDevice(), &semaphore_create_info,
                           nullptr, &render_finished[i]) != VK_SUCCESS) ||
        (vkCreateFence(device->getLogicalDevice(), &fence_create_info, nullptr,
                       &in_flight_fences[i]) != VK_SUCCESS)) {
      throw std::runtime_error("Failed to create a semaphore and/or fence!");
    }
  }
}

void VulkanRenderer::create_uniform_buffers() {
  // one uniform buffer for each image (and by extension, command buffer)
  globalUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());
  sceneUBOBuffer.resize(vulkanSwapChain.getNumberSwapChainImages());

  //// temporary buffer to "stage" vertex data before transfering to GPU
  // VulkanBuffer       stagingBuffer;
  std::vector<GlobalUBO> globalUBOdata;
  globalUBOdata.push_back(globalUBO);

  std::vector<SceneUBO> sceneUBOdata;
  sceneUBOdata.push_back(sceneUBO);

  // create uniform buffers
  for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
    vulkanBufferManager.createBufferAndUploadVectorOnDevice(
        device.get(), graphics_command_pool, globalUBOBuffer[i],
        VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, globalUBOdata);

    vulkanBufferManager.createBufferAndUploadVectorOnDevice(
        device.get(), graphics_command_pool, sceneUBOBuffer[i],
        VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, sceneUBOdata);
  }
}

void VulkanRenderer::createDescriptorPoolSharedRenderStages() {
  // CREATE UNIFORM DESCRIPTOR POOL
  // type of descriptors + how many descriptors, not descriptor sets (combined
  // makes the pool size) ViewProjection Pool
  VkDescriptorPoolSize vp_pool_size{};
  vp_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
  vp_pool_size.descriptorCount = static_cast<uint32_t>(globalUBOBuffer.size());

  // DIRECTION POOL
  VkDescriptorPoolSize directions_pool_size{};
  directions_pool_size.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
  directions_pool_size.descriptorCount =
      static_cast<uint32_t>(sceneUBOBuffer.size());

  VkDescriptorPoolSize object_descriptions_pool_size{};
  object_descriptions_pool_size.type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
  object_descriptions_pool_size.descriptorCount =
      static_cast<uint32_t>(sizeof(ObjectDescription) * MAX_OBJECTS);

  // TEXTURE SAMPLER POOL
  VkDescriptorPoolSize sampler_pool_size{};
  sampler_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLER;
  sampler_pool_size.descriptorCount = MAX_TEXTURE_COUNT;

  VkDescriptorPoolSize sampled_image_pool_size{};
  sampled_image_pool_size.type = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
  sampled_image_pool_size.descriptorCount = MAX_TEXTURE_COUNT;

  // list of pool sizes
  std::vector<VkDescriptorPoolSize> descriptor_pool_sizes = {
      vp_pool_size, directions_pool_size, object_descriptions_pool_size,
      sampler_pool_size, sampled_image_pool_size};

  VkDescriptorPoolCreateInfo pool_create_info{};
  pool_create_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
  pool_create_info.maxSets =
      vulkanSwapChain
          .getNumberSwapChainImages();  // maximum number of descriptor sets
                                        // that can be created from pool
  pool_create_info.poolSizeCount = static_cast<uint32_t>(
      descriptor_pool_sizes.size());  // amount of pool sizes being passed
  pool_create_info.pPoolSizes =
      descriptor_pool_sizes.data();  // pool sizes to create pool with

  // create descriptor pool
  VkResult result =
      vkCreateDescriptorPool(device->getLogicalDevice(), &pool_create_info,
                             nullptr, &descriptorPoolSharedRenderStages);
  ASSERT_VULKAN(result, "Failed to create a descriptor pool!")
}

void VulkanRenderer::createSharedRenderDescriptorSet() {
  // resize descriptor set list so one for every buffer
  sharedRenderDescriptorSet.resize(vulkanSwapChain.getNumberSwapChainImages());

  std::vector<VkDescriptorSetLayout> set_layouts(
      vulkanSwapChain.getNumberSwapChainImages(),
      sharedRenderDescriptorSetLayout);

  // descriptor set allocation info
  VkDescriptorSetAllocateInfo set_alloc_info{};
  set_alloc_info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
  set_alloc_info.descriptorPool =
      descriptorPoolSharedRenderStages;  // pool to allocate descriptor set from
  set_alloc_info.descriptorSetCount =
      vulkanSwapChain.getNumberSwapChainImages();  // number of sets to allocate
  set_alloc_info.pSetLayouts =
      set_layouts.data();  // layouts to use to allocate sets (1:1 relationship)

  // allocate descriptor sets (multiple)
  VkResult result =
      vkAllocateDescriptorSets(device->getLogicalDevice(), &set_alloc_info,
                               sharedRenderDescriptorSet.data());
  ASSERT_VULKAN(result, "Failed to create descriptor sets!")

  // update all of descriptor set buffer bindings
  for (size_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
    // VIEW PROJECTION DESCRIPTOR
    // buffer info and data offset info
    VkDescriptorBufferInfo globalUBO_buffer_info{};
    globalUBO_buffer_info.buffer =
        globalUBOBuffer[i].getBuffer();  // buffer to get data from
    globalUBO_buffer_info.offset = 0;    // position of start of data
    globalUBO_buffer_info.range = sizeof(globalUBO);  // size of data

    // data about connection between binding and buffer
    VkWriteDescriptorSet globalUBO_set_write{};
    globalUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    globalUBO_set_write.dstSet =
        sharedRenderDescriptorSet[i];  // descriptor set to update
    globalUBO_set_write.dstBinding =
        0;  // binding to update (matches with binding on layout/shader)
    globalUBO_set_write.dstArrayElement = 0;  // index in array to update
    globalUBO_set_write.descriptorType =
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;    // type of descriptor
    globalUBO_set_write.descriptorCount = 1;  // amount to update
    globalUBO_set_write.pBufferInfo =
        &globalUBO_buffer_info;  // information about buffer data to bind

    // VIEW PROJECTION DESCRIPTOR
    // buffer info and data offset info
    VkDescriptorBufferInfo sceneUBO_buffer_info{};
    sceneUBO_buffer_info.buffer =
        sceneUBOBuffer[i].getBuffer();              // buffer to get data from
    sceneUBO_buffer_info.offset = 0;                // position of start of data
    sceneUBO_buffer_info.range = sizeof(sceneUBO);  // size of data

    // data about connection between binding and buffer
    VkWriteDescriptorSet sceneUBO_set_write{};
    sceneUBO_set_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    sceneUBO_set_write.dstSet =
        sharedRenderDescriptorSet[i];  // descriptor set to update
    sceneUBO_set_write.dstBinding =
        1;  // binding to update (matches with binding on layout/shader)
    sceneUBO_set_write.dstArrayElement = 0;  // index in array to update
    sceneUBO_set_write.descriptorType =
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;   // type of descriptor
    sceneUBO_set_write.descriptorCount = 1;  // amount to update
    sceneUBO_set_write.pBufferInfo =
        &sceneUBO_buffer_info;  // information about buffer data to bind

    std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
        globalUBO_set_write, sceneUBO_set_write};

    // update the descriptor sets with new buffer/binding info
    vkUpdateDescriptorSets(device->getLogicalDevice(),
                           static_cast<uint32_t>(write_descriptor_sets.size()),
                           write_descriptor_sets.data(), 0, nullptr);
  }
}

void VulkanRenderer::updateTexturesInSharedRenderDescriptorSet() {
  std::vector<Texture>& modelTextures = scene->getTextures(0);
  std::vector<VkDescriptorImageInfo> image_info_textures;
  image_info_textures.resize(scene->getTextureCount(0));
  for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
    image_info_textures[i].imageLayout =
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    image_info_textures[i].imageView = modelTextures[i].getImageView();
    image_info_textures[i].sampler = nullptr;
  }

  std::vector<VkSampler>& modelTextureSampler = scene->getTextureSampler(0);
  std::vector<VkDescriptorImageInfo> image_info_texture_sampler;
  image_info_texture_sampler.resize(scene->getTextureCount(0));
  for (uint32_t i = 0; i < scene->getTextureCount(0); i++) {
    image_info_texture_sampler[i].imageLayout =
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    image_info_texture_sampler[i].imageView = nullptr;
    image_info_texture_sampler[i].sampler = modelTextureSampler[i];
  }

  for (uint32_t i = 0; i < vulkanSwapChain.getNumberSwapChainImages(); i++) {
    // descriptor write info
    VkWriteDescriptorSet descriptor_write{};
    descriptor_write.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    descriptor_write.dstSet = sharedRenderDescriptorSet[i];
    descriptor_write.dstBinding = TEXTURES_BINDING;
    descriptor_write.dstArrayElement = 0;
    descriptor_write.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
    descriptor_write.descriptorCount =
        static_cast<uint32_t>(image_info_textures.size());
    descriptor_write.pImageInfo = image_info_textures.data();

    /*VkDescriptorImageInfo sampler_info;
                sampler_info.imageView = nullptr;
                sampler_info.sampler = texture_sampler;*/

    // descriptor write info
    VkWriteDescriptorSet descriptor_write_sampler{};
    descriptor_write_sampler.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    descriptor_write_sampler.dstSet = sharedRenderDescriptorSet[i];
    descriptor_write_sampler.dstBinding = SAMPLER_BINDING;
    descriptor_write_sampler.dstArrayElement = 0;
    descriptor_write_sampler.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
    descriptor_write_sampler.descriptorCount =
        static_cast<uint32_t>(image_info_texture_sampler.size());
    descriptor_write_sampler.pImageInfo = image_info_texture_sampler.data();

    std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
        descriptor_write, descriptor_write_sampler};

    // update new descriptor set
    vkUpdateDescriptorSets(device->getLogicalDevice(),
                           static_cast<uint32_t>(write_descriptor_sets.size()),
                           write_descriptor_sets.data(), 0, nullptr);
  }
}

void VulkanRenderer::cleanUpUBOs() {
  for (VulkanBuffer vulkanBuffer : globalUBOBuffer) {
    vulkanBuffer.cleanUp();
  }

  for (VulkanBuffer vulkanBuffer : sceneUBOBuffer) {
    vulkanBuffer.cleanUp();
  }
}

void VulkanRenderer::update_uniform_buffers(uint32_t image_index) {
  auto usage_stage_flags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
                           VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR |
                           VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

  VkBufferMemoryBarrier before_barrier_uvp{};
  before_barrier_uvp.pNext = nullptr;
  before_barrier_uvp.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
  before_barrier_uvp.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
  before_barrier_uvp.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  before_barrier_uvp.buffer = globalUBOBuffer[image_index].getBuffer();
  before_barrier_uvp.offset = 0;
  before_barrier_uvp.size = sizeof(globalUBO);
  before_barrier_uvp.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  before_barrier_uvp.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

  VkBufferMemoryBarrier before_barrier_directions{};
  before_barrier_directions.pNext = nullptr;
  before_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
  before_barrier_directions.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
  before_barrier_directions.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  before_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
  before_barrier_directions.offset = 0;
  before_barrier_directions.size = sizeof(sceneUBO);
  before_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  before_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

  vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
                       VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
                       &before_barrier_uvp, 0, nullptr);
  vkCmdPipelineBarrier(command_buffers[image_index], usage_stage_flags,
                       VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 1,
                       &before_barrier_directions, 0, nullptr);

  vkCmdUpdateBuffer(command_buffers[image_index],
                    globalUBOBuffer[image_index].getBuffer(), 0,
                    sizeof(GlobalUBO), &globalUBO);
  vkCmdUpdateBuffer(command_buffers[image_index],
                    sceneUBOBuffer[image_index].getBuffer(), 0,
                    sizeof(SceneUBO), &sceneUBO);

  VkBufferMemoryBarrier after_barrier_uvp{};
  after_barrier_uvp.pNext = nullptr;
  after_barrier_uvp.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
  after_barrier_uvp.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  after_barrier_uvp.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
  after_barrier_uvp.buffer = globalUBOBuffer[image_index].getBuffer();
  after_barrier_uvp.offset = 0;
  after_barrier_uvp.size = sizeof(GlobalUBO);
  after_barrier_uvp.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  after_barrier_uvp.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

  VkBufferMemoryBarrier after_barrier_directions{};
  after_barrier_directions.pNext = nullptr;
  after_barrier_directions.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
  after_barrier_directions.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  after_barrier_directions.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
  after_barrier_directions.buffer = globalUBOBuffer[image_index].getBuffer();
  after_barrier_directions.offset = 0;
  after_barrier_directions.size = sizeof(SceneUBO);
  after_barrier_directions.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  after_barrier_directions.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

  vkCmdPipelineBarrier(command_buffers[image_index],
                       VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
                       nullptr, 1, &after_barrier_uvp, 0, nullptr);
  vkCmdPipelineBarrier(command_buffers[image_index],
                       VK_PIPELINE_STAGE_TRANSFER_BIT, usage_stage_flags, 0, 0,
                       nullptr, 1, &after_barrier_directions, 0, nullptr);
}

void VulkanRenderer::update_raytracing_descriptor_set(uint32_t image_index) {
  VkWriteDescriptorSetAccelerationStructureKHR
      descriptor_set_acceleration_structure{};
  descriptor_set_acceleration_structure.sType =
      VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
  descriptor_set_acceleration_structure.pNext = nullptr;
  descriptor_set_acceleration_structure.accelerationStructureCount = 1;
  VkAccelerationStructureKHR& tlasAS = asManager.getTLAS();
  descriptor_set_acceleration_structure.pAccelerationStructures = &tlasAS;

  VkWriteDescriptorSet write_descriptor_set_acceleration_structure{};
  write_descriptor_set_acceleration_structure.sType =
      VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
  write_descriptor_set_acceleration_structure.pNext =
      &descriptor_set_acceleration_structure;
  write_descriptor_set_acceleration_structure.dstSet =
      raytracingDescriptorSet[image_index];
  write_descriptor_set_acceleration_structure.dstBinding = TLAS_BINDING;
  write_descriptor_set_acceleration_structure.dstArrayElement = 0;
  write_descriptor_set_acceleration_structure.descriptorCount = 1;
  write_descriptor_set_acceleration_structure.descriptorType =
      VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
  write_descriptor_set_acceleration_structure.pImageInfo = nullptr;
  write_descriptor_set_acceleration_structure.pBufferInfo = nullptr;
  write_descriptor_set_acceleration_structure.pTexelBufferView = nullptr;

  VkDescriptorBufferInfo object_description_buffer_info{};
  object_description_buffer_info.buffer = objectDescriptionBuffer.getBuffer();
  object_description_buffer_info.offset = 0;
  object_description_buffer_info.range = VK_WHOLE_SIZE;

  VkWriteDescriptorSet object_description_buffer_write{};
  object_description_buffer_write.sType =
      VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
  object_description_buffer_write.dstSet =
      sharedRenderDescriptorSet[image_index];
  object_description_buffer_write.descriptorType =
      VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
  object_description_buffer_write.dstBinding = OBJECT_DESCRIPTION_BINDING;
  object_description_buffer_write.pBufferInfo = &object_description_buffer_info;
  object_description_buffer_write.descriptorCount = 1;

  std::vector<VkWriteDescriptorSet> write_descriptor_sets = {
      write_descriptor_set_acceleration_structure,
      object_description_buffer_write};

  vkUpdateDescriptorSets(device->getLogicalDevice(),
                         static_cast<uint32_t>(write_descriptor_sets.size()),
                         write_descriptor_sets.data(), 0, nullptr);
}

void VulkanRenderer::record_commands(uint32_t image_index) {
  Texture& renderResult = rasterizer.getOffscreenTexture(image_index);
  VulkanImage& vulkanImage = renderResult.getVulkanImage();

  GUIRendererSharedVars& guiRendererSharedVars =
      gui->getGuiRendererSharedVars();
  if (guiRendererSharedVars.raytracing) {
    std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],
                                         raytracingDescriptorSet[image_index]};
    raytracingStage.recordCommands(command_buffers[image_index],
                                   &vulkanSwapChain, sets);

  } else if (guiRendererSharedVars.pathTracing) {
    std::vector<VkDescriptorSet> sets = {sharedRenderDescriptorSet[image_index],
                                         raytracingDescriptorSet[image_index]};

    pathTracing.recordCommands(command_buffers[image_index], image_index,
                               vulkanImage, &vulkanSwapChain, sets);

  } else {
    std::vector<VkDescriptorSet> descriptorSets = {
        sharedRenderDescriptorSet[image_index]};

    rasterizer.recordCommands(command_buffers[image_index], image_index, scene,
                              descriptorSets);
  }

  vulkanImage.transitionImageLayout(
      command_buffers[image_index], VK_IMAGE_LAYOUT_GENERAL,
      VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);

  std::vector<VkDescriptorSet> descriptorSets = {
      post_descriptor_set[image_index]};
  postStage.recordCommands(command_buffers[image_index], image_index,
                           descriptorSets);

  vulkanImage.transitionImageLayout(
      command_buffers[image_index], VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
      VK_IMAGE_LAYOUT_GENERAL, 1, VK_IMAGE_ASPECT_COLOR_BIT);
}

bool VulkanRenderer::checkChangedFramebufferSize() {
  if (window->framebuffer_size_has_changed()) {
    vkDeviceWaitIdle(device->getLogicalDevice());
    vkQueueWaitIdle(device->getGraphicsQueue());

    vulkanSwapChain.cleanUp();
    vulkanSwapChain.initVulkanContext(device.get(), window, surface);

    std::vector<VkDescriptorSetLayout> descriptor_set_layouts = {
        sharedRenderDescriptorSetLayout};
    rasterizer.cleanUp();
    rasterizer.init(device.get(), &vulkanSwapChain, descriptor_set_layouts,
                    graphics_command_pool);

    // all post
    std::vector<VkDescriptorSetLayout> descriptorSets = {
        post_descriptor_set_layout};
    postStage.cleanUp();
    postStage.init(device.get(), &vulkanSwapChain, descriptorSets);

    gui->cleanUp();
    gui->initializeVulkanContext(device.get(), instance.getVulkanInstance(),
                                 postStage.getRenderPass(),
                                 graphics_command_pool);

    current_frame = 0;

    updatePostDescriptorSets();
    updateRaytracingDescriptorSets();

    window->reset_framebuffer_has_changed();

    return true;
  }

  return false;
}

void VulkanRenderer::cleanUp() {
  cleanUpUBOs();

  rasterizer.cleanUp();
  raytracingStage.cleanUp();
  postStage.cleanUp();
  pathTracing.cleanUp();

  objectDescriptionBuffer.cleanUp();
  asManager.cleanUp();

  vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
                               raytracingDescriptorSetLayout, nullptr);
  vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
                               post_descriptor_set_layout, nullptr);
  vkDestroyDescriptorSetLayout(device->getLogicalDevice(),
                               sharedRenderDescriptorSetLayout, nullptr);
  vkDestroyDescriptorPool(device->getLogicalDevice(), post_descriptor_pool,
                          nullptr);
  vkDestroyDescriptorPool(device->getLogicalDevice(),
                          descriptorPoolSharedRenderStages, nullptr);
  vkDestroyDescriptorPool(device->getLogicalDevice(), raytracingDescriptorPool,
                          nullptr);

  vkFreeCommandBuffers(device->getLogicalDevice(), graphics_command_pool,
                       static_cast<uint32_t>(command_buffers.size()),
                       command_buffers.data());

  cleanUpCommandPools();

  cleanUpSync();

  vulkanSwapChain.cleanUp();
  vkDestroySurfaceKHR(instance.getVulkanInstance(), surface, nullptr);
  allocator.cleanUp();
  device->cleanUp();
  debug::freeDebugCallback(instance.getVulkanInstance());
  instance.cleanUp();
}

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.

```
#include "Camera.h"

Camera::Camera()
    :

      position(glm::vec3(0.0f, 100.0f, -80.0f)),
      front(glm::vec3(0.0f, 0.0f, -1.f)),
      world_up(glm::vec3(0.0f, 1.0f, 0.0f)),
      right(glm::normalize(glm::cross(front, world_up))),
      up(glm::normalize(glm::cross(right, front))),
      yaw(80.f),
      pitch(-40.0f),
      movement_speed(200.f),
      turn_speed(0.25f),
      near_plane(0.1f),
      far_plane(4000.f),
      fov(45.f)

{}

void Camera::key_control(bool* keys, float delta_time) {
  float velocity = movement_speed * delta_time;

  if (keys[GLFW_KEY_W]) {
    position += front * velocity;
  }

  if (keys[GLFW_KEY_D]) {
    position += right * velocity;
  }

  if (keys[GLFW_KEY_A]) {
    position += -right * velocity;
  }

  if (keys[GLFW_KEY_S]) {
    position += -front * velocity;
  }

  if (keys[GLFW_KEY_Q]) {
    yaw += -velocity;
  }

  if (keys[GLFW_KEY_E]) {
    yaw += velocity;
  }
}

void Camera::mouse_control(float x_change, float y_change) {
  // here we only want to support views 90 degrees to each side
  // again choose turn speed well in respect to its ordinal scale
  x_change *= turn_speed;
  y_change *= turn_speed;

  yaw += x_change;
  pitch += y_change;

  if (pitch > 89.0f) {
    pitch = 89.0f;
  }

  if (pitch < -89.0f) {
    pitch = -89.0f;
  }

  // by changing the rotations you need to update all parameters
  // for we retrieve them later for further calculations!
  update();
}

void Camera::set_near_plane(float near_plane) { this->near_plane = near_plane; }

void Camera::set_far_plane(float far_plane) { this->far_plane = far_plane; }

void Camera::set_fov(float fov) { this->fov = fov; }

void Camera::set_camera_position(glm::vec3 new_camera_position) {
  this->position = new_camera_position;
}

glm::mat4 Camera::calculate_viewmatrix() {
  // very necessary for further calc
  return glm::lookAt(position, position + front, up);
}

Camera::~Camera() {}

void Camera::update() {
  //
 https://learnopengl.com/Getting-started/Camera?fbclid=IwAR1WEr4jt6IyWC52s_WKYHtaFoeug37pG5YqbDPifgn5F1UXPbUjWbJWiqQ
  //  thats a bit tricky; have a look to link above if there a questions :)
  //  but simple geometrical analysis
  //  consider yaw you are turnig to the side; pich as you move the head forward
  //  and back; roll rotations around z-axis will make you dizzy :)) notice that
  //  to roll will not chnge my front vector
  front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
  front.y = sin(glm::radians(pitch));
  front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
  front = glm::normalize(front);

  // retrieve the right vector with some world_up
  right = glm::normalize(glm::cross(front, world_up));

  // but this means the up vector must again be calculated with right vector
  // calculated!!!
  up = glm::normalize(glm::cross(right, front));
}
```

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.

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

```

#include <cstring>
#include <memory>

#include "VulkanBuffer.h"

Mesh::Mesh() {}

void Mesh::cleanUp() {
  vertexBuffer.cleanUp();
  indexBuffer.cleanUp();
  objectDescriptionBuffer.cleanUp();
  materialIdsBuffer.cleanUp();
  materialsBuffer.cleanUp();
}

Mesh::Mesh(VulkanDevice* device, VkQueue transfer_queue,
           VkCommandPool transfer_command_pool, std::vector<Vertex>& vertices,
           std::vector<uint32_t>& indices,
           std::vector<unsigned int>& materialIndex,
           std::vector<ObjMaterial>& materials) {
  // glm uses column major matrices so transpose it for Vulkan want row major
  // here
  glm::mat4 transpose_transform = glm::transpose(glm::mat4(1.0f));
  VkTransformMatrixKHR out_matrix;
  std::memcpy(&out_matrix, &transpose_transform, sizeof(VkTransformMatrixKHR));

  index_count = static_cast<uint32_t>(indices.size());
  vertex_count = static_cast<uint32_t>(vertices.size());
  this->device = device;
  object_description = ObjectDescription{};
  createVertexBuffer(transfer_queue, transfer_command_pool, vertices);
  createIndexBuffer(transfer_queue, transfer_command_pool, indices);
  createMaterialIDBuffer(transfer_queue, transfer_command_pool, materialIndex);
  createMaterialBuffer(transfer_queue, transfer_command_pool, materials);

  VkBufferDeviceAddressInfo vertex_info{};
  vertex_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
  vertex_info.buffer = vertexBuffer.getBuffer();

  VkBufferDeviceAddressInfo index_info{};
  index_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
  index_info.buffer = indexBuffer.getBuffer();

  VkBufferDeviceAddressInfo material_index_info{};
  material_index_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
  material_index_info.buffer = materialIdsBuffer.getBuffer();

  VkBufferDeviceAddressInfo material_info{};
  material_info.sType = VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR;
  material_info.buffer = materialsBuffer.getBuffer();

  object_description.index_address =
      vkGetBufferDeviceAddress(device->getLogicalDevice(), &index_info);
  object_description.vertex_address =
      vkGetBufferDeviceAddress(device->getLogicalDevice(), &vertex_info);
  object_description.material_index_address = vkGetBufferDeviceAddress(
      device->getLogicalDevice(), &material_index_info);
  object_description.material_address =
      vkGetBufferDeviceAddress(device->getLogicalDevice(), &material_info);

  model = glm::mat4(1.0f);
}

void Mesh::setModel(glm::mat4 new_model) { model = new_model; }

Mesh::~Mesh() {}

void Mesh::createVertexBuffer(VkQueue transfer_queue,
                              VkCommandPool transfer_command_pool,
                              std::vector<Vertex>& vertices) {
  vulkanBufferManager.createBufferAndUploadVectorOnDevice(
      device, transfer_command_pool, vertexBuffer,
      VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT |
          VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
          VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
          VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
      VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
          VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
      vertices);
}

void Mesh::createIndexBuffer(VkQueue transfer_queue,
                             VkCommandPool transfer_command_pool,
                             std::vector<uint32_t>& indices) {
  vulkanBufferManager.createBufferAndUploadVectorOnDevice(
      device, transfer_command_pool, indexBuffer,
      VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
          VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
          VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
          VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
      VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
          VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
      indices);
}

void Mesh::createMaterialIDBuffer(VkQueue transfer_queue,
                                  VkCommandPool transfer_command_pool,
                                  std::vector<unsigned int>& materialIndex) {
  vulkanBufferManager.createBufferAndUploadVectorOnDevice(
      device, transfer_command_pool, materialIdsBuffer,
      VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
          VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
          VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
          VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
      VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
          VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
      materialIndex);
}

void Mesh::createMaterialBuffer(VkQueue transfer_queue,
                                VkCommandPool transfer_command_pool,
                                std::vector<ObjMaterial>& materials) {
  vulkanBufferManager.createBufferAndUploadVectorOnDevice(
      device, transfer_command_pool, materialsBuffer,
      VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT |
          VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
          VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
          VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
      VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
          VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
      materials);
}
```

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.

```
#include "Model.h"

Model::Model() {}

Model::Model(VulkanDevice* device) { this->device = device; }

void Model::cleanUp() {
  for (Texture texture : modelTextures) {
    texture.cleanUp();
  }

  for (VkSampler texture_sampler : modelTextureSamplers) {
    vkDestroySampler(device->getLogicalDevice(), texture_sampler, nullptr);
  }

  mesh.cleanUp();
}

void Model::add_new_mesh(VulkanDevice* device, VkQueue transfer_queue,
                         VkCommandPool command_pool,
                         std::vector<Vertex>& vertices,
                         std::vector<unsigned int>& indices,
                         std::vector<unsigned int>& materialIndex,
                         std::vector<ObjMaterial>& materials) {
  this->mesh = Mesh(device, transfer_queue, command_pool, vertices, indices,
                    materialIndex, materials);
}
```

```

void Model::set_model(glm::mat4 model) { this->model = model; }

void Model::addTexture(Texture newTexture) {
  modelTextures.push_back(newTexture);
  addSampler(newTexture);
}

uint32_t Model::getPrimitiveCount() {
  /*uint32_t number_of_indices = 0;

    for (Mesh mesh : meshes) {

        number_of_indices += mesh.get_index_count();

    }

    return number_of_indices / 3;*/
  return mesh.getIndexCount() / 3;
}

Model::~Model() {}

void Model::addSampler(Texture newTexture) {
  VkSampler newSampler;
  // sampler create info
  VkSamplerCreateInfo sampler_create_info{};
  sampler_create_info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
  sampler_create_info.magFilter = VK_FILTER_LINEAR;
  sampler_create_info.minFilter = VK_FILTER_LINEAR;
  sampler_create_info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  sampler_create_info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  sampler_create_info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  sampler_create_info.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
  sampler_create_info.unnormalizedCoordinates = VK_FALSE;
  sampler_create_info.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
  sampler_create_info.mipLodBias = 0.0f;
  sampler_create_info.minLod = 0.0f;
  sampler_create_info.maxLod = newTexture.getMipLevel();
  sampler_create_info.anisotropyEnable = VK_TRUE;
  sampler_create_info.maxAnisotropy = 16;  // max anisotropy sample level

  VkResult result = vkCreateSampler(device->getLogicalDevice(),
                                    &sampler_create_info, nullptr, &newSampler);
  ASSERT_VULKAN(result, "Failed to create a texture sampler!")

  modelTextureSamplers.push_back(newSampler);
}
```

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];
```

```
    ObjMaterial material{};
    material.ambient =
        glm::vec3(mp->ambient[0], mp->ambient[1], mp->ambient[2]);
    material.diffuse =
        glm::vec3(mp->diffuse[0], mp->diffuse[1], mp->diffuse[2]);
    material.specular =
        glm::vec3(mp->specular[0], mp->specular[1], mp->specular[2]);
    material.emission =
        glm::vec3(mp->emission[0], mp->emission[1], mp->emission[2]);
    material.transmittance = glm::vec3(
        mp->transmittance[0], mp->transmittance[1], mp->transmittance[2]);
    material.dissolve = mp->dissolve;
    material.ior = mp->ior;
    material.shininess = mp->shininess;
    material.illum = mp->illum;

    if (mp->diffuse_texname.length() > 0) {
      std::string relative_texture_filename = mp->diffuse_texname;
      File model_file(modelFile);
      std::string texture_filename =
          model_file.getBaseDir() + "/textures/" + relative_texture_filename;

      textures.push_back(texture_filename);
      material.textureID = texture_id;
      texture_id++;

    } else {
      material.textureID = 0;
      textures.push_back("");
    }

    materials.push_back(material);
  }

  // for the case no .mtl file is given place some random standard material ...
  if (tol_materials.empty()) {
    materials.emplace_back(ObjMaterial());
  }

  return textures;
}

void ObjLoader::loadVertices(const std::string& fileName) {
  tinyobj::ObjReaderConfig reader_config;
  // reader_config.mtl_search_path = ""; // Path to material files

  tinyobj::ObjReader reader;

  if (!reader.ParseFromFile(fileName, reader_config)) {
    if (!reader.Error().empty()) {
      std::cerr « "TinyObjReader: " « reader.Error();
    }
    exit(EXIT_FAILURE);
  }

  if (!reader.Warning().empty()) {
    std::cout « "TinyObjReader: " « reader.Warning();
  }

  auto& attrib = reader.GetAttrib();
  auto& shapes = reader.GetShapes();
  auto& materials = reader.GetMaterials();

  std::unordered_map<Vertex, uint32_t> vertices_map{};

  // Loop over shapes
  for (size_t s = 0; s < shapes.size(); s++) {
    // prepare for enlargement
    vertices.reserve(shapes[s].mesh.indices.size() + vertices.size());
    indices.reserve(shapes[s].mesh.indices.size() + indices.size());

    // Loop over faces(polygon)
    size_t index_offset = 0;
    for (size_t f = 0; f < shapes[s].mesh.num_face_vertices.size(); f++) {
      size_t fv = size_t(shapes[s].mesh.num_face_vertices[f]);

      // Loop over vertices in the face.
      for (size_t v = 0; v < fv; v++) {
        // access to vertex
        tinyobj::index_t idx = shapes[s].mesh.indices[index_offset + v];
        tinyobj::real_t vx = attrib.vertices[3 * size_t(idx.vertex_index) + 0];
        tinyobj::real_t vy = attrib.vertices[3 * size_t(idx.vertex_index) + 1];
        tinyobj::real_t vz = attrib.vertices[3 * size_t(idx.vertex_index) + 2];
        glm::vec3 pos = {vx, vy, vz};

        glm::vec3 normals(0.0f);
        // Check if `normal_index` is zero or positive. negative = no normal
        // data
        if (idx.normal_index >= 0 && !attrib.normals.empty()) {
          tinyobj::real_t nx = attrib.normals[3 * size_t(idx.normal_index) + 0];
          tinyobj::real_t ny = attrib.normals[3 * size_t(idx.normal_index) + 1];
          tinyobj::real_t nz = attrib.normals[3 * size_t(idx.normal_index) + 2];
          normals = glm::vec3(nx, ny, nz);
        }

        glm::vec3 color(-1.f);
        if (!attrib.colors.empty()) {
          tinyobj::real_t red = attrib.colors[3 * size_t(idx.vertex_index) + 0];
          tinyobj::real_t green =
              attrib.colors[3 * size_t(idx.vertex_index) + 1];
          tinyobj::real_t blue =
              attrib.colors[3 * size_t(idx.vertex_index) + 2];
          color = glm::vec3(red, green, blue);
        }

        glm::vec2 tex_coords(0.0f);
        // Check if `texcoord_index` is zero or positive. negative = no texcoord
        // data
        if (idx.texcoord_index >= 0 && !attrib.texcoords.empty()) {
          tinyobj::real_t tx =
              attrib.texcoords[2 * size_t(idx.texcoord_index) + 0];
          // flip y coordinate !!
          tinyobj::real_t ty =
              1.f - attrib.texcoords[2 * size_t(idx.texcoord_index) + 1];
          tex_coords = glm::vec2(tx, ty);
        }

        Vertex vert{pos, normals, color, tex_coords};

        if (vertices_map.count(vert) == 0) {
          vertices_map[vert] = vertices.size();
          vertices.push_back(vert);
        }

        indices.push_back(vertices_map[vert]);
      }

      index_offset += fv;

      // per-face material; face usually is triangle
      // matToTex[shapes[s].mesh.material_ids[f]]
      materialIndex.push_back(shapes[s].mesh.material_ids[f]);
    }
  }

  // precompute normals if no provided
  if (attrib.normals.empty()) {
    for (size_t i = 0; i < indices.size(); i += 3) {
      Vertex& v0 = vertices[indices[i + 0]];
      Vertex& v1 = vertices[indices[i + 1]];
      Vertex& v2 = vertices[indices[i + 2]];

      glm::vec3 n =
          glm::normalize(glm::cross((v1.pos - v0.pos), (v2.pos - v0.pos)));
      v0.normal = n;
      v1.normal = n;
      v2.normal = n;
    }
  }
}
```

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.

```
#include "Scene.h"

Scene::Scene() {}

void Scene::update_user_input(GUI* gui) {
  guiSceneSharedVars = gui->getGuiSceneSharedVars();
}

void Scene::loadModel(VulkanDevice* device, VkCommandPool commandPool) {
  ObjLoader obj_loader(device, device->getGraphicsQueue(), commandPool);

  std::string modelFileName = sceneConfig::getModelFile();
  std::shared_ptr<Model> new_model = obj_loader.loadModel(modelFileName);

  add_model(new_model);

  glm::mat4 modelMatrix = sceneConfig::getModelMatrix();

  update_model_matrix(modelMatrix, 0);
}

void Scene::add_model(std::shared_ptr<Model> model) {
  model_list.push_back(model);
  object_descriptions.push_back(model->getObjectDescription());
}

void Scene::add_object_description(ObjectDescription object_description) {
  object_descriptions.push_back(object_description);
}

void Scene::update_model_matrix(glm::mat4 model_matrix, int model_id) {
  if (model_id >= static_cast<int32_t>(getModelCount()) || model_id < 0) {
    throw std::runtime_error("Wrong model id value!");
  }

  model_list[model_id]->set_model(model_matrix);
}

void Scene::cleanUp() {
  for (std::shared_ptr<Model> model : model_list) {
    model->cleanUp();
  }
}

uint32_t Scene::getNumberMeshes() {
  uint32_t number_of_meshes = 0;

  for (std::shared_ptr<Model> mesh_model : model_list) {
    number_of_meshes += static_cast<uint32_t>(mesh_model->getMeshCount());
  }

  return number_of_meshes;
}

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.

```
#include <SceneConfig.h>

//#define SULO_MODE 0

namespace sceneConfig {

std::string getModelFile() {
  std::stringstream modelFile;
  modelFile « CMAKELISTS_DIR;
#if NDEBUG
  modelFile « "/Resources/Model/crytek-sponza/";
  modelFile « "sponza_triag.obj";

#else
#ifdef SULO_MODE
  modelFile « "/Resources/Model/Sulo/";
  modelFile « "SuloLongDongLampe_v2.obj";
#else
  modelFile « "/Resources/Model/VikingRoom/";
  modelFile « "viking_room.obj";
#endif
#endif

  return modelFile.str();
  // std::string modelFile =
  // "../Resources/Model/crytek-sponza/sponza_triag.obj"; std::string modelFile
  // = "../Resources/Model/Dinosaurs/dinosaurs.obj"; std::string modelFile =
  // "../Resources/Model/Pillum/PilumPainting_export.obj"; std::string modelFile
  // = "../Resources/Model/sibenik/sibenik.obj"; std::string modelFile =
  // "../Resources/Model/sportsCar/sportsCar.obj"; std::string modelFile =
  // "../Resources/Model/StanfordDragon/dragon.obj"; std::string modelFile =
  // "../Resources/Model/CornellBox/CornellBox-Sphere.obj"; std::string
  // modelFile = "../Resources/Model/bunny/bunny.obj"; std::string modelFile =
  // "../Resources/Model/buddha/buddha.obj"; std::string modelFile =
  // "../Resources/Model/bmw/bmw.obj"; std::string modelFile =
  // "../Resources/Model/testScene.obj"; std::string modelFile =
  // "../Resources/Model/San_Miguel/san-miguel-low-poly.obj";
}

glm::mat4 getModelMatrix() {
  glm::mat4 modelMatrix(1.0f);

#if NDEBUG

  // dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
  // -50.0f));
  modelMatrix = glm::scale(modelMatrix, glm::vec3(1.0f, 1.0f, 1.0f));
  /*dragon_model = glm::rotate(dragon_model, glm::radians(-90.f),
     glm::vec3(1.0f, 0.0f, 0.0f)); dragon_model = glm::rotate(dragon_model,
     glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));*/

#else

// dragon_model = glm::translate(dragon_model, glm::vec3(0.0f, -40.0f,
// -50.0f));
#if SULO_MODE
  modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
#else
  modelMatrix = glm::scale(modelMatrix, glm::vec3(60.0f, 60.0f, 60.0f));
  modelMatrix = glm::rotate(modelMatrix, glm::radians(-90.f),
                            glm::vec3(1.0f, 0.0f, 0.0f));
  modelMatrix =
      glm::rotate(modelMatrix, glm::radians(90.f), glm::vec3(0.0f, 0.0f, 1.0f));
#endif

#endif

  return modelMatrix;
}

}  // 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.

```
#include "Texture.h"

#include <cmath>
#include <stdexcept>

Texture::Texture() {}

void Texture::createFromFile(VulkanDevice* device, VkCommandPool commandPool,
                             const std::string& fileName) {
  int width, height;
  VkDeviceSize size;
  stbi_uc* image_data = loadTextureData(fileName, &width, &height, &size);

  mip_levels =
      static_cast<uint32_t>(std::floor(std::log2(std::max(width, height)))) + 1;

  // create staging buffer to hold loaded data, ready to copy to device
  VulkanBuffer stagingBuffer;
  stagingBuffer.create(device, size, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
                       VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
                           VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

  // copy image data to staging buffer
  void* data;
  vkMapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory(), 0,
              size, 0, &data);
  memcpy(data, image_data, static_cast<size_t>(size));
  vkUnmapMemory(device->getLogicalDevice(), stagingBuffer.getBufferMemory());

  // free original image data
  stbi_image_free(image_data);

  createImage(device, width, height, mip_levels, VK_FORMAT_R8G8B8A8_UNORM,
              VK_IMAGE_TILING_OPTIMAL,
              VK_IMAGE_USAGE_TRANSFER_SRC_BIT |
                  VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
              VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

  // copy data to image
  // transition image to be DST for copy operation
  vulkanImage.transitionImageLayout(
      device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
      VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
      VK_IMAGE_ASPECT_COLOR_BIT, mip_levels);

  // copy data to image
  vulkanBufferManager.copyImageBuffer(
      device->getLogicalDevice(), device->getGraphicsQueue(), commandPool,
      stagingBuffer.getBuffer(), vulkanImage.getImage(), width, height);

  // generate mipmaps
  generateMipMaps(device->getPhysicalDevice(), device->getLogicalDevice(),
                  commandPool, device->getGraphicsQueue(),
                  vulkanImage.getImage(), VK_FORMAT_R8G8B8A8_SRGB, width,
                  height, mip_levels);

  stagingBuffer.cleanUp();

  createImageView(device, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_COLOR_BIT,
                  mip_levels);
}

void Texture::setImage(VkImage image) { vulkanImage.setImage(image); }

void Texture::setImageView(VkImageView imageView) {
  vulkanImageView.setImageView(imageView);
}

void Texture::createImage(VulkanDevice* device, uint32_t width, uint32_t height,
                          uint32_t mip_levels, VkFormat format,
                          VkImageTiling tiling, VkImageUsageFlags use_flags,
                          VkMemoryPropertyFlags prop_flags) {
  vulkanImage.create(device, width, height, mip_levels, format, tiling,
                     use_flags, prop_flags);
}
```

```

void Texture::createImageView(VulkanDevice* device, VkFormat format,
                              VkImageAspectFlags aspect_flags,
                              uint32_t mip_levels) {
  vulkanImageView.create(device, vulkanImage.getImage(), format, aspect_flags,
                         mip_levels);
}

void Texture::cleanUp() {
  vulkanImageView.cleanUp();
  vulkanImage.cleanUp();
}

Texture::~Texture() {}

stbi_uc* Texture::loadTextureData(const std::string& file_name, int* width,
                                  int* height, VkDeviceSize* image_size) {
  // number of channels image uses
  int channels;
  // load pixel data for image
  // std::string file_loc = "../Resources/Textures/" + file_name;
  stbi_uc* image =
      stbi_load(file_name.c_str(), width, height, &channels, STBI_rgb_alpha);

  if (!image) {
    throw std::runtime_error("Failed to load a texture file! (" + file_name +
                             ")");
  }

  // calculate image size using given and known data
  *image_size = *width * *height * 4;

  return image;
}

void Texture::generateMipMaps(VkPhysicalDevice physical_device, VkDevice device,
                              VkCommandPool command_pool, VkQueue queue,
                              VkImage image, VkFormat image_format,
                              int32_t width, int32_t height,
                              uint32_t mip_levels) {
  // Check if image format supports linear blitting
  VkFormatProperties formatProperties;
  vkGetPhysicalDeviceFormatProperties(physical_device, image_format,
                                      &formatProperties);

  if (!(formatProperties.optimalTilingFeatures &
        VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT)) {
    throw std::runtime_error(
        "Texture image format does not support linear blitting!");
  }

  VkCommandBuffer command_buffer =
      commandBufferManager.beginCommandBuffer(device, command_pool);

  VkImageMemoryBarrier barrier{};
  barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
  barrier.image = image;
  barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
  barrier.subresourceRange.baseArrayLayer = 0;
  barrier.subresourceRange.layerCount = 1;
  barrier.subresourceRange.levelCount = 1;

  // TEMP VARS needed for decreasing step by step for factor 2
  int32_t tmp_width = width;
  int32_t tmp_height = height;

  // -- WE START AT 1 !
  for (uint32_t i = 1; i < mip_levels; i++) {
    // WAIT for previous mip map level for being ready
    barrier.subresourceRange.baseMipLevel = i - 1;
    // HERE we TRANSITION for having a SRC format now
    barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    barrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;

    vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
                         VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 0,
                         nullptr, 1, &barrier);

    // when barrier over we can now blit :)
    VkImageBlit blit{};

    // -- OFFSETS describing the 3D-dimesnion of the region
    blit.srcOffsets[0] = {0, 0, 0};
    blit.srcOffsets[1] = {tmp_width, tmp_height, 1};
    blit.srcSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    // copy from previous level
    blit.srcSubresource.mipLevel = i - 1;
    blit.srcSubresource.baseArrayLayer = 0;
    blit.srcSubresource.layerCount = 1;
    // -- OFFSETS describing the 3D-dimesnion of the region
    blit.dstOffsets[0] = {0, 0, 0};
    blit.dstOffsets[1] = {tmp_width > 1 ? tmp_width / 2 : 1,
                          tmp_height > 1 ? tmp_height / 2 : 1, 1};
    blit.dstSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    // -- COPY to next mipmap level
    blit.dstSubresource.mipLevel = i;
    blit.dstSubresource.baseArrayLayer = 0;
    blit.dstSubresource.layerCount = 1;

    vkCmdBlitImage(command_buffer, image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
                   image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &blit,
                   VK_FILTER_LINEAR);

    // REARRANGE image formats for having the correct image formats again
    barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

    vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
                         VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, nullptr,
                         0, nullptr, 1, &barrier);

    if (tmp_width > 1) tmp_width /= 2;
    if (tmp_height > 1) tmp_height /= 2;
  }

  barrier.subresourceRange.baseMipLevel = mip_levels - 1;
  barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
  barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
  barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

  vkCmdPipelineBarrier(command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
                       VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, nullptr, 0,
                       nullptr, 1, &barrier);

  commandBufferManager.endAndSubmitCommandBuffer(device, command_pool, queue,
                                                 command_buffer);
}
```

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.

```
#include "Vertex.h"

Vertex::Vertex() {
  this->pos = glm::vec3(-1.f);
  this->normal = glm::vec3(-1.f);
  this->color = glm::vec3(-1.f);
  this->texture_coords = glm::vec3(-1.f);
}

Vertex::Vertex(glm::vec3 pos, glm::vec3 normal, glm::vec3 color,
               glm::vec2 texture_coords) {
  this->pos = pos;
  this->normal = normal;
  this->color = color;
  this->texture_coords = texture_coords;
}

namespace vertex {

std::array<VkVertexInputAttributeDescription, 4> getVertexInputAttributeDesc() {
  std::array<VkVertexInputAttributeDescription, 4> attribute_describtions;

  // Position attribute
  attribute_describtions[0].binding = 0;
  attribute_describtions[0].location = 0;
  attribute_describtions[0].format =
      VK_FORMAT_R32G32B32_SFLOAT;  // format data will take (also helps define
                                   // size of data)
  attribute_describtions[0].offset = offsetof(Vertex, pos);

  // normal coord attribute
  attribute_describtions[1].binding = 0;
  attribute_describtions[1].location = 1;
  attribute_describtions[1].format =
      VK_FORMAT_R32G32B32_SFLOAT;  // format data will take (also helps define
                                   // size of data)
  attribute_describtions[1].offset =
      offsetof(Vertex, normal);  // where this attribute is defined in the data
                                 // for a single vertex

  // normal coord attribute
  attribute_describtions[2].binding = 0;
  attribute_describtions[2].location = 2;
  attribute_describtions[2].format =
      VK_FORMAT_R32G32B32_SFLOAT;  // format data will take (also helps define
                                   // size of data)
  attribute_describtions[2].offset = offsetof(Vertex, color);

  attribute_describtions[3].binding = 0;
  // texture coord attribute
  attribute_describtions[3].location = 3;
  attribute_describtions[3].format =
      VK_FORMAT_R32G32_SFLOAT;  // format data will take (also helps define size
                                // of data)
  attribute_describtions[3].offset =
      offsetof(Vertex, texture_coords);  // where this attribute is defined in
                                         // the data for a single vertex

  return attribute_describtions;
}

}  // 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.

```
#include "File.h"

#include <fstream>
#include <iostream>

File::File(const std::string& file_location) {
  this->file_location = file_location;
}

std::string File::read() {
  std::string content;
  std::ifstream file_stream(file_location, std::ios::in);

  if (!file_stream.is_open()) {
    printf("Failed to read %s. File does not exist.", file_location.c_str());
    return "";
  }

  std::string line = "";
  while (!file_stream.eof()) {
    std::getline(file_stream, line);
    content.append(line + "\n");
  }

  file_stream.close();
  return content;
}

std::vector<char> File::readCharSequence() {
  // open stream from given file
  // std::ios::binary tells stream to read file as binary
  // std::ios:ate tells stream to start reading from end of file
  std::ifstream file(file_location, std::ios::binary | std::ios::ate);

  // check if file stream sucessfully opened
  if (!file.is_open()) {
    throw std::runtime_error("Failed to open a file!");
  }

  size_t file_size = (size_t)file.tellg();
  std::vector<char> file_buffer(file_size);

  // move read position to start of file
  file.seekg(0);

  // read the file data into the buffer (stream "file_size" in total)
  file.read(file_buffer.data(), file_size);

  file.close();

  return file_buffer;
}

std::string File::getBaseDir() {
  if (file_location.find_last_of("/\\") != std::string::npos)
    return file_location.substr(0, file_location.find_last_of("/\\"));
  return "";
}

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.

```
#include "ShaderHelper.h"

#include <sstream>

#include "Utilities.h"

ShaderHelper::ShaderHelper() {}

void ShaderHelper::compileShader(const std::string& shader_src_dir,
                                 const std::string& shader_name) {
  // GLSLC_EXE is set by cmake to the location of the vulkan glslc
  std::stringstream shader_src_path;
  std::stringstream shader_log_file;
  std::stringstream cmdShaderCompile;
  std::stringstream adminPriviliges;
  adminPriviliges « "runas /user:<admin-user> \"";

  shader_src_path « shader_src_dir « shader_name;
  std::string shader_spv_path = getShaderSpvDir(shader_src_dir, shader_name);
  shader_log_file « shader_src_dir « shader_name « ".log.txt";
  std::stringstream log_stdout_and_stderr;
  log_stdout_and_stderr « " > " « shader_log_file.str() « " 2> "
                        « shader_log_file.str();

  cmdShaderCompile  //« adminPriviliges.str()
      « GLSLC_EXE « target « shader_src_path.str() « " -o "
      « shader_spv_path;
  //« log_stdout_and_stderr.str();

  // std::cout « cmdShaderCompile.str().c_str();

  system(cmdShaderCompile.str().c_str());
}

std::string ShaderHelper::getShaderSpvDir(const std::string& shader_src_dir,
                                          const std::string& shader_name) {
  std::string shader_spv_dir = "spv/";

  std::stringstream vertShaderSpv;
  vertShaderSpv « shader_src_dir « shader_spv_dir « shader_name « ".spv";

  return vertShaderSpv.str();
}

VkShaderModule ShaderHelper::createShaderModule(VulkanDevice* device,
                                                const std::vector<char>& code) {
  // shader module create info
  VkShaderModuleCreateInfo shader_module_create_info{};
  shader_module_create_info.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
  shader_module_create_info.codeSize = code.size();  // size of code
  shader_module_create_info.pCode =
      reinterpret_cast<const uint32_t*>(code.data());  // pointer to code

  VkShaderModule shader_module;
  VkResult result =
      vkCreateShaderModule(device->getLogicalDevice(),
                           &shader_module_create_info, nullptr, &shader_module);

  ASSERT_VULKAN(result, "Failed to create a shader module!")

  return shader_module;
}

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.

```
#include "VulkanBuffer.h"

#include <stdexcept>

#include "MemoryHelper.h"
#include "Utilities.h"

VulkanBuffer::VulkanBuffer() {}

void VulkanBuffer::create(VulkanDevice* device, VkDeviceSize buffer_size,
                          VkBufferUsageFlags buffer_usage_flags,
                          VkMemoryPropertyFlags buffer_propertiy_flags) {
  this->device = device;

  // information to create a buffer (doesn't include assigning memory)
  VkBufferCreateInfo buffer_info{};
  buffer_info.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
  buffer_info.size = buffer_size;
  // multiple types of buffer possible, e.g. vertex buffer
  buffer_info.usage = buffer_usage_flags;
  // similar to swap chain images, can share vertex buffers
  buffer_info.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

  VkResult result = vkCreateBuffer(device->getLogicalDevice(), &buffer_info,
                                   nullptr, &buffer);
  ASSERT_VULKAN(result, "Failed to create a buffer!");

  // get buffer memory requirements
  VkMemoryRequirements memory_requirements{};
  vkGetBufferMemoryRequirements(device->getLogicalDevice(), buffer,
                                &memory_requirements);

  // allocate memory to buffer
  VkMemoryAllocateInfo memory_alloc_info{};
  memory_alloc_info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
  memory_alloc_info.allocationSize = memory_requirements.size;

  uint32_t memory_type_index = find_memory_type_index(
      device->getPhysicalDevice(), memory_requirements.memoryTypeBits,
      buffer_propertiy_flags);

  // VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |              /* memory is visible to
  // CPU side
  // */ VK_MEMORY_PROPERTY_HOST_COHERENT_BIT    /* data is placed straight into
  // buffer */);
  if (memory_type_index < 0) {
    throw std::runtime_error("Failed to find suitable memory type!");
  }

  memory_alloc_info.memoryTypeIndex = memory_type_index;

  // allocate memory to VkDeviceMemory
  result = vkAllocateMemory(device->getLogicalDevice(), &memory_alloc_info,
                            nullptr, &bufferMemory);
  ASSERT_VULKAN(result, "Failed to allocate memory for buffer!");

  // allocate memory to given buffer
  vkBindBufferMemory(device->getLogicalDevice(), buffer, bufferMemory, 0);

  created = true;
}

void VulkanBuffer::cleanUp() {
  if (created) {
    vkDestroyBuffer(device->getLogicalDevice(), buffer, nullptr);
    vkFreeMemory(device->getLogicalDevice(), bufferMemory, nullptr);
  }
}

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.

```
#include "VulkanBufferManager.h"

VulkanBufferManager::VulkanBufferManager() {}

void VulkanBufferManager::copyBuffer(VkDevice device, VkQueue transfer_queue,
                                     VkCommandPool transfer_command_pool,
                                     VulkanBuffer src_buffer,
                                     VulkanBuffer dst_buffer,
                                     VkDeviceSize buffer_size) {
  // create buffer
  VkCommandBuffer command_buffer =
      commandBufferManager.beginCommandBuffer(device, transfer_command_pool);

  // region of data to copy from and to
  VkBufferCopy buffer_copy_region{};
  buffer_copy_region.srcOffset = 0;
  buffer_copy_region.dstOffset = 0;
  buffer_copy_region.size = buffer_size;

  // command to copy src buffer to dst buffer
  vkCmdCopyBuffer(command_buffer, src_buffer.getBuffer(),
                  dst_buffer.getBuffer(), 1, &buffer_copy_region);

  commandBufferManager.endAndSubmitCommandBuffer(
      device, transfer_command_pool, transfer_queue, command_buffer);
}

void VulkanBufferManager::copyImageBuffer(VkDevice device,
                                          VkQueue transfer_queue,
                                          VkCommandPool transfer_command_pool,
                                          VkBuffer src_buffer, VkImage image,
                                          uint32_t width, uint32_t height) {
  // create buffer
  VkCommandBuffer transfer_command_buffer =
      commandBufferManager.beginCommandBuffer(device, transfer_command_pool);

  VkBufferImageCopy image_region{};
  image_region.bufferOffset = 0;  // offset into data
  image_region.bufferRowLength =
      0;  // row length of data to calculate data spacing
  image_region.bufferImageHeight = 0;  // image height to calculate data spacing
  image_region.imageSubresource.aspectMask =
      VK_IMAGE_ASPECT_COLOR_BIT;  // which aspect of image to copy
  image_region.imageSubresource.mipLevel = 0;
  image_region.imageSubresource.baseArrayLayer = 0;
  image_region.imageSubresource.layerCount = 1;
  image_region.imageOffset = {0, 0, 0};  // offset into image
  image_region.imageExtent = {width, height, 1};

  // copy buffer to given image
  vkCmdCopyBufferToImage(transfer_command_buffer, src_buffer, image,
                         VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1,
                         &image_region);

  commandBufferManager.endAndSubmitCommandBuffer(
      device, transfer_command_pool, transfer_queue, transfer_command_buffer);
}

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 (VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity, VkDebugUtilsMessageTypeFlagsEXT messageType, const VkDebugUtilsMessengerCallbackDataEXT ∗pCallbackData, void ∗pUserData)
- void debug::setupDebugging (VkInstance instance, VkDebugReportFlagsEXT flags, VkDebugReportCallbackEXT callBack)
- void debug::freeDebugCallback (VkInstance instance)

Variables

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

4.48 VulkanDebug.cpp

Go to the documentation of this file.

```
#include "VulkanDebug.h"

#include "Utilities.h"

namespace debug {
PFN_vkCreateDebugUtilsMessengerEXT vkCreateDebugUtilsMessengerEXT;
PFN_vkDestroyDebugUtilsMessengerEXT vkDestroyDebugUtilsMessengerEXT;
VkDebugUtilsMessengerEXT debugUtilsMessenger;

VKAPI_ATTR VkBool32 VKAPI_CALL debugUtilsMessengerCallback(
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageType,
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
    void* pUserData) {
  // Select prefix depending on flags passed to the callback
  std::string prefix("");

  if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT) {
    prefix = "VERBOSE: ";
  } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT) {
    prefix = "INFO: ";
  } else if (messageSeverity &
             VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
    prefix = "WARNING: ";
  } else if (messageSeverity & VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
    prefix = "ERROR: ";
  }

  // Display message to default output (console/logcat)
  std::stringstream debugMessage;
  debugMessage « prefix « "[" « pCallbackData->messageIdNumber « "]["
               « pCallbackData->pMessageIdName
               « "] : " « pCallbackData->pMessage;

#if defined(__ANDROID__)
  if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
    LOGE("%s", debugMessage.str().c_str());
  } else {
    LOGD("%s", debugMessage.str().c_str());
  }
#else
  if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT) {
    std::cerr « debugMessage.str() « "\n";
  } else {
    std::cout « debugMessage.str() « "\n";
  }
  fflush(stdout);
#endif

  // The return value of this callback controls whether the Vulkan call that
  // caused the validation message will be aborted or not We return VK_FALSE as
  // we DON'T want Vulkan calls that cause a validation message to abort If you
  // instead want to have calls abort, pass in VK_TRUE and the function will
  // return VK_ERROR_VALIDATION_FAILED_EXT
  return VK_FALSE;
}

void setupDebugging(VkInstance instance, VkDebugReportFlagsEXT flags,
                    VkDebugReportCallbackEXT callBack) {
  vkCreateDebugUtilsMessengerEXT =
      reinterpret_cast<PFN_vkCreateDebugUtilsMessengerEXT>(
          vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT"));
  vkDestroyDebugUtilsMessengerEXT =
      reinterpret_cast<PFN_vkDestroyDebugUtilsMessengerEXT>(
          vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT"));

  VkDebugUtilsMessengerCreateInfoEXT debugUtilsMessengerCI{};
  debugUtilsMessengerCI.sType =
      VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
  debugUtilsMessengerCI.messageSeverity =
      VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
      VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
  debugUtilsMessengerCI.messageType =
      VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
      VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT;
  debugUtilsMessengerCI.pfnUserCallback = debugUtilsMessengerCallback;
  ASSERT_VULKAN(vkCreateDebugUtilsMessengerEXT(instance, &debugUtilsMessengerCI,
                                               nullptr, &debugUtilsMessenger),
                "Failed to create debug messenger")
}

void freeDebugCallback(VkInstance instance) {
  if (debugUtilsMessenger != VK_NULL_HANDLE) {
    vkDestroyDebugUtilsMessengerEXT(instance, debugUtilsMessenger, nullptr);
  }
}
}  // 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.

```
#include "VulkanDevice.h"

#include <string.h>

#include <set>
#include <string>

VulkanDevice::VulkanDevice(VulkanInstance* instance, VkSurfaceKHR* surface) {
  this->instance = instance;
  this->surface = surface;
  get_physical_device();
  create_logical_device();
}

SwapChainDetails VulkanDevice::getSwapchainDetails() {
  return getSwapchainDetails(physical_device);
}

void VulkanDevice::cleanUp() { vkDestroyDevice(logical_device, nullptr); }

VulkanDevice::~VulkanDevice() {}

QueueFamilyIndices VulkanDevice::getQueueFamilies() {
  QueueFamilyIndices indices{};

  uint32_t queue_family_count = 0;
  vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
                                           nullptr);

  std::vector<VkQueueFamilyProperties> queue_family_list(queue_family_count);
  vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
                                           queue_family_list.data());

  // Go through each queue family and check if it has at least 1 of required
  // types we need to keep track th eindex by our own
  int index = 0;
  for (const auto& queue_family : queue_family_list) {
    // first check if queue family has at least 1 queue in that family
    // Queue can be multiple types defined through bitfield. Need to bitwise AND
    // with VK_QUE_*_BIT to check if has required  type
    if (queue_family.queueCount > 0 &&
        queue_family.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
      indices.graphics_family = index;  // if queue family valid, than get index
    }

    if (queue_family.queueCount > 0 &&
        queue_family.queueFlags & VK_QUEUE_COMPUTE_BIT) {
      indices.compute_family = index;
    }

    // check if queue family suppports presentation
    VkBool32 presentation_support = false;
    vkGetPhysicalDeviceSurfaceSupportKHR(physical_device, index, *surface,
                                         &presentation_support);
    // check if queue is presentation type (can be both graphics and
    // presentation)
    if (queue_family.queueCount > 0 && presentation_support) {
      indices.presentation_family = index;
    }

    // check if queue family indices are in a valid state
    if (indices.is_valid()) {
      break;
    }

    index++;
  }

  return indices;
}

void VulkanDevice::get_physical_device() {
  // Enumerate physical devices the vkInstance can access
  uint32_t device_count = 0;
  vkEnumeratePhysicalDevices(instance->getVulkanInstance(), &device_count,
                             nullptr);

  // if no devices available, then none support of Vulkan
  if (device_count == 0) {
    throw std::runtime_error(
        "Can not find GPU's that support Vulkan Instance!");
  }

  // Get list of physical devices
  std::vector<VkPhysicalDevice> device_list(device_count);
  vkEnumeratePhysicalDevices(instance->getVulkanInstance(), &device_count,
                             device_list.data());

  for (const auto& device : device_list) {
    if (check_device_suitable(device)) {
      physical_device = device;
      break;
    }
  }

  // get properties of our new device
  vkGetPhysicalDeviceProperties(physical_device, &device_properties);
}

void VulkanDevice::create_logical_device() {
  // get the queue family indices for the chosen physical device
  QueueFamilyIndices indices = getQueueFamilies();

  // vector for queue creation information and set for family indices
  std::vector<VkDeviceQueueCreateInfo> queue_create_infos;
  std::set<int> queue_family_indices = {indices.graphics_family,
                                        indices.presentation_family,
                                        indices.compute_family};

  // Queue the logical device needs to create and info to do so (only 1 for now,
  // will add more later!)
  for (int queue_family_index : queue_family_indices) {
    VkDeviceQueueCreateInfo queue_create_info{};
    queue_create_info.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queue_create_info.queueFamilyIndex =
        queue_family_index;  // the index of the family to create a queue from
    queue_create_info.queueCount = 1;  // number of queues to create
    float priority = 1.0f;
    queue_create_info.pQueuePriorities =
        &priority;  // Vulkan needs to know how to handle multiple queues, so
                    // decide priority (1 = highest)

    queue_create_infos.push_back(queue_create_info);
  }

  // -- ALL EXTENSION WE NEED
  VkPhysicalDeviceDescriptorIndexingFeatures indexing_features{};
  indexing_features.sType =
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES;
  indexing_features.runtimeDescriptorArray = VK_TRUE;
  indexing_features.shaderSampledImageArrayNonUniformIndexing = VK_TRUE;
  indexing_features.pNext = nullptr;

  // -- NEEDED FOR QUERING THE DEVICE ADDRESS WHEN CREATING ACCELERATION
  // STRUCTURES
  VkPhysicalDeviceBufferDeviceAddressFeaturesEXT
      buffer_device_address_features{};
  buffer_device_address_features.sType =
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT;
  buffer_device_address_features.pNext = &indexing_features;
  buffer_device_address_features.bufferDeviceAddress = VK_TRUE;
  buffer_device_address_features.bufferDeviceAddressCaptureReplay = VK_TRUE;
  buffer_device_address_features.bufferDeviceAddressMultiDevice = VK_FALSE;

  // --ENABLE RAY TRACING PIPELINE
  VkPhysicalDeviceRayTracingPipelineFeaturesKHR ray_tracing_pipeline_features{};
  ray_tracing_pipeline_features.sType =
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_FEATURES_KHR;
  ray_tracing_pipeline_features.pNext = &buffer_device_address_features;
  ray_tracing_pipeline_features.rayTracingPipeline = VK_TRUE;

  // -- ENABLE ACCELERATION STRUCTURES
  VkPhysicalDeviceAccelerationStructureFeaturesKHR
      acceleration_structure_features{};
  acceleration_structure_features.sType =
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_FEATURES_KHR;
  acceleration_structure_features.pNext = &ray_tracing_pipeline_features;
  acceleration_structure_features.accelerationStructure = VK_TRUE;
  acceleration_structure_features.accelerationStructureCaptureReplay = VK_TRUE;
  acceleration_structure_features.accelerationStructureIndirectBuild = VK_FALSE;
  acceleration_structure_features.accelerationStructureHostCommands = VK_FALSE;
  acceleration_structure_features
      .descriptorBindingAccelerationStructureUpdateAfterBind = VK_FALSE;

  VkPhysicalDeviceVulkan13Features features13{};
  features13.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES;
  features13.maintenance4 = VK_TRUE;
  features13.robustImageAccess = VK_FALSE;
  features13.inlineUniformBlock = VK_FALSE;
  features13.descriptorBindingInlineUniformBlockUpdateAfterBind = VK_FALSE;
  features13.pipelineCreationCacheControl = VK_FALSE;
  features13.privateData = VK_FALSE;
  features13.shaderDemoteToHelperInvocation = VK_FALSE;
  features13.shaderTerminateInvocation = VK_FALSE;
  features13.subgroupSizeControl = VK_FALSE;
  features13.computeFullSubgroups = VK_FALSE;
  features13.synchronization2 = VK_FALSE;
  features13.textureCompressionASTC_HDR = VK_FALSE;
  features13.shaderZeroInitializeWorkgroupMemory = VK_FALSE;
  features13.dynamicRendering = VK_FALSE;
  features13.shaderIntegerDotProduct = VK_FALSE;
  features13.pNext = &acceleration_structure_features;

  VkPhysicalDeviceRayQueryFeaturesKHR rayQueryFeature{};
  rayQueryFeature.sType =
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_QUERY_FEATURES_KHR;
  rayQueryFeature.pNext = &features13;
  rayQueryFeature.rayQuery = VK_TRUE;

  VkPhysicalDeviceFeatures2 features2{};
  features2.pNext = &rayQueryFeature;
  features2.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2;
  features2.features.samplerAnisotropy = VK_TRUE;
  features2.features.shaderInt64 = VK_TRUE;
  features2.features.geometryShader = VK_TRUE;
  features2.features.logicOp = VK_TRUE;

  // -- PREPARE FOR HAVING MORE EXTENSION BECAUSE WE NEED RAYTRACING
  // CAPABILITIES
  std::vector<const char*> extensions(device_extensions);

  // COPY ALL NECESSARY EXTENSIONS FOR RAYTRACING TO THE EXTENSION
  extensions.insert(extensions.begin(),
                    device_extensions_for_raytracing.begin(),
                    device_extensions_for_raytracing.end());

  // information to create logical device (sometimes called "device")
  VkDeviceCreateInfo device_create_info{};
  device_create_info.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
  device_create_info.queueCreateInfoCount = static_cast<uint32_t>(
      queue_create_infos.size());  // number of queue create infos
  device_create_info.pQueueCreateInfos =
      queue_create_infos.data();  // list of queue create infos so device can
                                  // create required queues
  device_create_info.enabledExtensionCount = static_cast<uint32_t>(
      extensions.size());  // number of enabled logical device extensions
  device_create_info.ppEnabledExtensionNames =
      extensions.data();  // list of enabled logical device extensions
  device_create_info.flags = 0;
  device_create_info.pEnabledFeatures = NULL;

  device_create_info.pNext = &features2;

  // create logical device for the given physical device
  VkResult result = vkCreateDevice(physical_device, &device_create_info,
                                   nullptr, &logical_device);
  ASSERT_VULKAN(result, "Failed to create a logical device!");

  //  Queues are created at the same time as the device...
  // So we want handle to queues
  // From given logical device of given queue family, of given queue index (0
  // since only one queue), place reference in given VkQueue
  vkGetDeviceQueue(logical_device, indices.graphics_family, 0, &graphics_queue);
  vkGetDeviceQueue(logical_device, indices.presentation_family, 0,
                   &presentation_queue);
  vkGetDeviceQueue(logical_device, indices.compute_family, 0, &compute_queue);
}

QueueFamilyIndices VulkanDevice::getQueueFamilies(
    VkPhysicalDevice physical_device) {
  QueueFamilyIndices indices{};

  uint32_t queue_family_count = 0;
  vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
                                           nullptr);

  std::vector<VkQueueFamilyProperties> queue_family_list(queue_family_count);
  vkGetPhysicalDeviceQueueFamilyProperties(physical_device, &queue_family_count,
                                           queue_family_list.data());

  // Go through each queue family and check if it has at least 1 of required
  // types we need to keep track th eindex by our own
  int index = 0;
  for (const auto& queue_family : queue_family_list) {
    // first check if queue family has at least 1 queue in that family
    // Queue can be multiple types defined through bitfield. Need to bitwise AND
    // with VK_QUE_*_BIT to check if has required  type
    if (queue_family.queueCount > 0 &&
        queue_family.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
      indices.graphics_family = index;  // if queue family valid, than get index
    }

    if (queue_family.queueCount > 0 &&
        queue_family.queueFlags & VK_QUEUE_COMPUTE_BIT) {
      indices.compute_family = index;
    }

    // check if queue family suppports presentation
    VkBool32 presentation_support = false;
    vkGetPhysicalDeviceSurfaceSupportKHR(physical_device, index, *surface,
                                         &presentation_support);
    // check if queue is presentation type (can be both graphics and
    // presentation)
    if (queue_family.queueCount > 0 && presentation_support) {
      indices.presentation_family = index;
    }

    // check if queue family indices are in a valid state
    if (indices.is_valid()) {
      break;
    }

    index++;
  }

  return indices;
}
```

```
SwapChainDetails VulkanDevice::getSwapchainDetails(VkPhysicalDevice device) {
  SwapChainDetails swapchain_details{};
  // get the surface capabilities for the given surface on the given physical
  // device
  vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
      device, *surface, &swapchain_details.surface_capabilities);

  uint32_t format_count = 0;
  vkGetPhysicalDeviceSurfaceFormatsKHR(device, *surface, &format_count,
                                       nullptr);

  // if formats returned, get list of formats
  if (format_count != 0) {
    swapchain_details.formats.resize(format_count);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, *surface, &format_count,
                                         swapchain_details.formats.data());
  }

  uint32_t presentation_count = 0;
  vkGetPhysicalDeviceSurfacePresentModesKHR(device, *surface,
                                            &presentation_count, nullptr);

  // if presentation modes returned, get list of presentation modes
  if (presentation_count > 0) {
    swapchain_details.presentation_mode.resize(presentation_count);
    vkGetPhysicalDeviceSurfacePresentModesKHR(
        device, *surface, &presentation_count,
        swapchain_details.presentation_mode.data());
  }

  return swapchain_details;
}

bool VulkanDevice::check_device_suitable(VkPhysicalDevice device) {
  // Information about device itself (ID, name, type, vendor, etc)
  VkPhysicalDeviceProperties device_properties;
  vkGetPhysicalDeviceProperties(device, &device_properties);

  VkPhysicalDeviceFeatures device_features;
  vkGetPhysicalDeviceFeatures(device, &device_features);

  QueueFamilyIndices indices = getQueueFamilies(device);

  bool extensions_supported = check_device_extension_support(device);

  bool swap_chain_valid = false;

  if (extensions_supported) {
    SwapChainDetails swap_chain_details = getSwapchainDetails(device);
    swap_chain_valid = !swap_chain_details.presentation_mode.empty() &&
                       !swap_chain_details.formats.empty();
  }

  return indices.is_valid() && extensions_supported && swap_chain_valid &&
         device_features.samplerAnisotropy;
}

bool VulkanDevice::check_device_extension_support(VkPhysicalDevice device) {
  uint32_t extension_count = 0;
  vkEnumerateDeviceExtensionProperties(device, nullptr, &extension_count,
                                       nullptr);

  if (extension_count == 0) {
    return false;
  }

  // populate list of extensions
  std::vector<VkExtensionProperties> extensions(extension_count);
  vkEnumerateDeviceExtensionProperties(device, nullptr, &extension_count,
                                       extensions.data());

  for (const auto& device_extension : device_extensions) {
    bool has_extension = false;

    for (const auto& extension : extensions) {
      if (strcmp(device_extension, extension.extensionName) == 0) {
        has_extension = true;
        break;
      }
    }

    if (!has_extension) {
      return false;
    }
  }

  return true;
}
```

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.

```
#include "VulkanImage.h"

#include "MemoryHelper.h"
#include "Utilities.h"

VulkanImage::VulkanImage() {}

void VulkanImage::create(VulkanDevice* device, uint32_t width, uint32_t height,
                         uint32_t mip_levels, VkFormat format,
                         VkImageTiling tiling, VkImageUsageFlags use_flags,
                         VkMemoryPropertyFlags prop_flags) {
  this->device = device;
  // CREATE image
  // image creation info
  VkImageCreateInfo image_create_info{};
  image_create_info.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
  image_create_info.imageType = VK_IMAGE_TYPE_2D;   // type of image (1D, 2D, 3D)
  image_create_info.extent.width = width;           // width if image extent
  image_create_info.extent.height = height;         // height if image extent
  image_create_info.extent.depth = 1;               // height if image extent
  image_create_info.mipLevels = mip_levels;         // number of mipmap levels
  image_create_info.arrayLayers = 1;  // number of levels in image array
  image_create_info.format = format;  // format type of image
  image_create_info.tiling =
      tiling;  // tiling of image ("arranged" for optimal reading)
  image_create_info.initialLayout =
      VK_IMAGE_LAYOUT_UNDEFINED;  // layout of image data on creation
  image_create_info.usage =
      use_flags;  // bit flags defining what image will be used for
  image_create_info.samples =
      VK_SAMPLE_COUNT_1_BIT;  // number of samples for multisampling
  image_create_info.sharingMode =
      VK_SHARING_MODE_EXCLUSIVE;  // whether image can be shared between queues

  VkResult result = vkCreateImage(device->getLogicalDevice(),
                                  &image_create_info, nullptr, &image);
  ASSERT_VULKAN(result, "Failed to create an image!")

  // CREATE memory for image
  // get memory requirements for a type of image
  VkMemoryRequirements memory_requirements;
  vkGetImageMemoryRequirements(device->getLogicalDevice(), image,
                               &memory_requirements);

  // allocate memory using image requirements and user defined properties
  VkMemoryAllocateInfo memory_alloc_info{};
  memory_alloc_info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
  memory_alloc_info.allocationSize = memory_requirements.size;
  memory_alloc_info.memoryTypeIndex =
      find_memory_type_index(device->getPhysicalDevice(),
                             memory_requirements.memoryTypeBits, prop_flags);

  result = vkAllocateMemory(device->getLogicalDevice(), &memory_alloc_info,
                            nullptr, &imageMemory);
  ASSERT_VULKAN(result, "Failed to allocate memory!")

  // connect memory to image
  vkBindImageMemory(device->getLogicalDevice(), image, imageMemory, 0);
```

```
}

void VulkanImage::transitionImageLayout(VkDevice device, VkQueue queue,
                                        VkCommandPool command_pool,
                                        VkImageLayout old_layout,
                                        VkImageLayout new_layout,
                                        VkImageAspectFlags aspectMask,
                                        uint32_t mip_levels) {
  VkCommandBuffer command_buffer =
      commandBufferManager.beginCommandBuffer(device, command_pool);

  // VK_IMAGE_ASPECT_COLOR_BIT
  VkImageMemoryBarrier memory_barrier{};
  memory_barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
  memory_barrier.oldLayout = old_layout;
  memory_barrier.newLayout = new_layout;
  memory_barrier.srcQueueFamilyIndex =
      VK_QUEUE_FAMILY_IGNORED;  // Queue family to transition from
  memory_barrier.dstQueueFamilyIndex =
      VK_QUEUE_FAMILY_IGNORED;  // Queue family to transition to
  memory_barrier.image =
      image;  // image being accessed and modified as part of barrier
  memory_barrier.subresourceRange.aspectMask =
      aspectMask;  // aspect of image being altered
  memory_barrier.subresourceRange.baseMipLevel =
      0;  // first mip level to start alterations on
  memory_barrier.subresourceRange.levelCount =
      mip_levels;  // number of mip levels to alter starting from baseMipLevel
  memory_barrier.subresourceRange.baseArrayLayer =
      0;  // first layer to start alterations on
  memory_barrier.subresourceRange.layerCount =
      1;  // number of layers to alter starting from baseArrayLayer

  // if transitioning from new image to image ready to receive data
  memory_barrier.srcAccessMask = accessFlagsForImageLayout(old_layout);
  memory_barrier.dstAccessMask = accessFlagsForImageLayout(new_layout);

  VkPipelineStageFlags src_stage = pipelineStageForLayout(old_layout);
  VkPipelineStageFlags dst_stage = pipelineStageForLayout(new_layout);

  vkCmdPipelineBarrier(

      command_buffer, src_stage,
      dst_stage,  // pipeline stages (match to src and dst accessmask)
      0,          // no dependency flags
      0,
      nullptr,  // memory barrier count + data
      0,
      nullptr,  // buffer memory barrier count + data
      1,
      &memory_barrier  // image memory barrier count + data

  );

  commandBufferManager.endAndSubmitCommandBuffer(device, command_pool, queue,
                                                 command_buffer);
}

void VulkanImage::transitionImageLayout(VkCommandBuffer command_buffer,
                                        VkImageLayout old_layout,
                                        VkImageLayout new_layout,
                                        uint32_t mip_levels,
                                        VkImageAspectFlags aspectMask) {
  VkImageMemoryBarrier memory_barrier{};
  memory_barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
  memory_barrier.oldLayout = old_layout;
  memory_barrier.newLayout = new_layout;
  memory_barrier.srcQueueFamilyIndex =
      VK_QUEUE_FAMILY_IGNORED;  // Queue family to transition from
  memory_barrier.dstQueueFamilyIndex =
      VK_QUEUE_FAMILY_IGNORED;  // Queue family to transition to
  memory_barrier.image =
      image;  // image being accessed and modified as part of barrier
  memory_barrier.subresourceRange.aspectMask =
      aspectMask;  // aspect of image being altered
  memory_barrier.subresourceRange.baseMipLevel =
      0;  // first mip level to start alterations on
  memory_barrier.subresourceRange.levelCount =
      mip_levels;  // number of mip levels to alter starting from baseMipLevel
  memory_barrier.subresourceRange.baseArrayLayer =
      0;  // first layer to start alterations on
  memory_barrier.subresourceRange.layerCount =
      1;  // number of layers to alter starting from baseArrayLayer

  memory_barrier.srcAccessMask = accessFlagsForImageLayout(old_layout);
  memory_barrier.dstAccessMask = accessFlagsForImageLayout(new_layout);

  VkPipelineStageFlags src_stage = pipelineStageForLayout(old_layout);
  VkPipelineStageFlags dst_stage = pipelineStageForLayout(new_layout);

  // if transitioning from new image to image ready to receive data

  vkCmdPipelineBarrier(

      command_buffer, src_stage,
      dst_stage,  // pipeline stages (match to src and dst accessmask)
      0,          // no dependency flags
      0,
      nullptr,  // memory barrier count + data
      0,
      nullptr,  // buffer memory barrier count + data
      1,
      &memory_barrier  // image memory barrier count + data

  );
}

void VulkanImage::setImage(VkImage image) { this->image = image; }

void VulkanImage::cleanUp() {
  vkDestroyImage(device->getLogicalDevice(), image, nullptr);
  vkFreeMemory(device->getLogicalDevice(), imageMemory, nullptr);
}

VulkanImage::~VulkanImage() {}

VkAccessFlags VulkanImage::accessFlagsForImageLayout(VkImageLayout layout) {
  switch (layout) {
    case VK_IMAGE_LAYOUT_PREINITIALIZED:
      return VK_ACCESS_HOST_WRITE_BIT;
    case VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:
      return VK_ACCESS_TRANSFER_WRITE_BIT;
    case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
      return VK_ACCESS_TRANSFER_READ_BIT;
    case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
      return VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
    case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
      return VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
      return VK_ACCESS_SHADER_READ_BIT;
    default:
      return VkAccessFlags();
  }
}

VkPipelineStageFlags VulkanImage::pipelineStageForLayout(
    VkImageLayout oldImageLayout) {
  switch (oldImageLayout) {
    case VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:
    case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
      return VK_PIPELINE_STAGE_TRANSFER_BIT;
    case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
      return VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
    case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
      return VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;  // We do this to allow queue
                                                  // other than graphic return
                                                  // VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
    case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
      return VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;  // We do this to allow queue
                                                  // other than graphic return
                                                  // VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
    case VK_IMAGE_LAYOUT_PREINITIALIZED:
      return VK_PIPELINE_STAGE_HOST_BIT;
    case VK_IMAGE_LAYOUT_UNDEFINED:
      return VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    default:
      return VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
  }
}
```

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.

```
#include "VulkanImageView.h"

VulkanImageView::VulkanImageView() {}

void VulkanImageView::setImageView(VkImageView imageView) {
  this->imageView = imageView;
}

void VulkanImageView::create(VulkanDevice* device, VkImage image,
                             VkFormat format, VkImageAspectFlags aspect_flags,
                             uint32_t mip_levels) {
  this->device = device;

  VkImageViewCreateInfo view_create_info{};
  view_create_info.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
  view_create_info.image = image;  // image to create view for
  view_create_info.viewType = VK_IMAGE_VIEW_TYPE_2D;  // typ of image
  view_create_info.format = format;
  view_create_info.components.r =
      VK_COMPONENT_SWIZZLE_IDENTITY;  // allows remapping of rgba components to
                                      // other rgba values
  view_create_info.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
  view_create_info.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
  view_create_info.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;

  // subresources allow the view to view only a part of an image
  view_create_info.subresourceRange.aspectMask =
      aspect_flags;  // which aspect of an image to view (e.g. color bit for
                     // viewing color)
  view_create_info.subresourceRange.baseMipLevel =
      0;  // start mipmap level to view from
  view_create_info.subresourceRange.levelCount =
      mip_levels;  // number of mipmap levels to view
  view_create_info.subresourceRange.baseArrayLayer =
      0;  // start array level to view from
  view_create_info.subresourceRange.layerCount =
      1;  // number of array levels to view

  // create image view
  VkResult result = vkCreateImageView(device->getLogicalDevice(),
                                      &view_create_info, nullptr, &imageView);
  ASSERT_VULKAN(result, "Failed to create an image view!")
}

void VulkanImageView::cleanUp() {
  vkDestroyImageView(device->getLogicalDevice(), imageView, nullptr);
}

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.

```
#include "VulkanInstance.h"

#include <string.h>

#include <string>

```

```
VulkanInstance::VulkanInstance() {
  if (ENABLE_VALIDATION_LAYERS && !check_validation_layer_support()) {
    throw std::runtime_error("Validation layers requested, but not available!");
  }

  // info about app
  // most data doesn’t affect program; is for developer convenience
  VkApplicationInfo app_info{};
  app_info.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
  app_info.pApplicationName =
      "\\__/ Epic Graphics from hell \\__/";  // custom name of app
  app_info.applicationVersion =
      VK_MAKE_VERSION(1, 3, 1);                       // custom version of app
  app_info.pEngineName = "Cataglyphis Renderer";      // custom engine name
  app_info.engineVersion = VK_MAKE_VERSION(1, 3, 3);  // custom engine version
  app_info.apiVersion = VK_API_VERSION_1_3;           // the vulkan version

  // creation info for a VkInstance
  VkInstanceCreateInfo create_info{};
  create_info.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
  create_info.pApplicationInfo = &app_info;

  // add validation layers IF enabled to the creeate info struct
  if (ENABLE_VALIDATION_LAYERS) {
    create_info.enabledLayerCount =
        static_cast<uint32_t>(validationLayers.size());
    create_info.ppEnabledLayerNames = validationLayers.data();

  } else {
    create_info.enabledLayerCount = 0;
    create_info.pNext = nullptr;
  }

  // create list to hold instance extensions
  std::vector<const char*> instance_extensions = std::vector<const char*>();

  // Setup extensions the instance will use
  uint32_t glfw_extensions_count = 0;  // GLFW may require multiple extensions
  const char** glfw_extensions;  // Extensions passed as array of cstrings, so
                                 // need pointer(array) to pointer

  // set GLFW extensions
  glfw_extensions = glfwGetRequiredInstanceExtensions(&glfw_extensions_count);

  // Add GLFW extensions to list of extensions
  for (size_t i = 0; i < glfw_extensions_count; i++) {
    instance_extensions.push_back(glfw_extensions[i]);
  }

  if (ENABLE_VALIDATION_LAYERS) {
    instance_extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
  }

  // check instance extensions supported
  if (!check_instance_extension_support(&instance_extensions)) {
    throw std::runtime_error(
        "VkInstance does not support required extensions!");
  }

  create_info.enabledExtensionCount =
      static_cast<uint32_t>(instance_extensions.size());
  create_info.ppEnabledExtensionNames = instance_extensions.data();

  // create instance
  VkResult result = vkCreateInstance(&create_info, nullptr, &instance);
  ASSERT_VULKAN(result, "Failed to create a Vulkan instance!");
}

bool VulkanInstance::check_validation_layer_support() {
  uint32_t layerCount;
  vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

  std::vector<VkLayerProperties> availableLayers(layerCount);
  vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

  for (const char* layerName : validationLayers) {
    bool layerFound = false;

    for (const auto& layerProperties : availableLayers) {
      if (strcmp(layerName, layerProperties.layerName) == 0) {
        layerFound = true;
        break;
      }
    }

    if (!layerFound) {
      return false;
    }
  }

  return true;
}

bool VulkanInstance::check_instance_extension_support(
    std::vector<const char*>* check_extensions) {
  // Need to get number of extensions to create array of correct size to hold
  // extensions
  uint32_t extension_count = 0;
  vkEnumerateInstanceExtensionProperties(nullptr, &extension_count, nullptr);

  // create a list of VkExtensionProperties using count
  std::vector<VkExtensionProperties> extensions(extension_count);
  vkEnumerateInstanceExtensionProperties(nullptr, &extension_count,
                                         extensions.data());

  // check if given extensions are in list of available extensions
  for (const auto& check_extension : *check_extensions) {
    bool has_extension = false;

    for (const auto& extension : extensions) {
      if (strcmp(check_extension, extension.extensionName)) {
        has_extension = true;
        break;
      }
    }

    if (!has_extension) {
      return false;
    }
  }

  return true;
}

void VulkanInstance::cleanUp() { vkDestroyInstance(instance, nullptr); }

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.

```
#include "VulkanSwapChain.h"

#include <limits>

#include "Utilities.h"

VulkanSwapChain::VulkanSwapChain() {}

void VulkanSwapChain::initVulkanContext(VulkanDevice* device, Window* window,
                                        const VkSurfaceKHR& surface) {
  this->device = device;
  this->window = window;

  // get swap chain details so we can pick the best settings
  SwapChainDetails swap_chain_details = device->getSwapchainDetails();

  // 1. choose best surface format
  // 2. choose best presentation mode
  // 3. choose swap chain image resolution

  VkSurfaceFormatKHR surface_format =
      choose_best_surface_format(swap_chain_details.formats);
  VkPresentModeKHR present_mode =
      choose_best_presentation_mode(swap_chain_details.presentation_mode);
  VkExtent2D extent =
      choose_swap_extent(swap_chain_details.surface_capabilities);

  // how many images are in the swap chain; get 1 more than the minimum to allow
  // tiple buffering
  uint32_t image_count =
      swap_chain_details.surface_capabilities.minImageCount + 1;

  // if maxImageCount == 0, then limitless
  if (swap_chain_details.surface_capabilities.maxImageCount > 0 &&
      swap_chain_details.surface_capabilities.maxImageCount < image_count) {
    image_count = swap_chain_details.surface_capabilities.maxImageCount;
  }

  VkSwapchainCreateInfoKHR swap_chain_create_info{};
  swap_chain_create_info.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
  swap_chain_create_info.surface = surface;  // swapchain surface
  swap_chain_create_info.imageFormat =
      surface_format.format;  // swapchain format
  swap_chain_create_info.imageColorSpace =
      surface_format.colorSpace;  // swapchain color space
  swap_chain_create_info.presentMode =
      present_mode;                             // swapchain presentation mode
  swap_chain_create_info.imageExtent = extent;  // swapchain image extents
  swap_chain_create_info.minImageCount =
      image_count;  // minimum images in swapchain
  swap_chain_create_info.imageArrayLayers =
      1;  // number of layers for each image in chain
  swap_chain_create_info.imageUsage =
      VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT |
      VK_IMAGE_USAGE_STORAGE_BIT |
      VK_IMAGE_USAGE_TRANSFER_DST_BIT;  // what attachment images will be used
                                        // as
  swap_chain_create_info.preTransform =
      swap_chain_details.surface_capabilities
          .currentTransform;  // transform to perform on swap chain images
  swap_chain_create_info.compositeAlpha =
      VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;  // dont do blending; everything opaque
  swap_chain_create_info.clipped = VK_TRUE;  // of course activate clipping ! :)

  // get queue family indices
  QueueFamilyIndices indices = device->getQueueFamilies();

  // if graphics and presentation families are different then swapchain must let
  // images be shared between families
  if (indices.graphics_family != indices.presentation_family) {
    uint32_t queue_family_indices[] = {(uint32_t)indices.graphics_family,
                                       (uint32_t)indices.presentation_family};

    swap_chain_create_info.imageSharingMode =
        VK_SHARING_MODE_CONCURRENT;  // image share handling
    swap_chain_create_info.queueFamilyIndexCount =
        2;  // number of queues to share images between
    swap_chain_create_info.pQueueFamilyIndices =
        queue_family_indices;  // array of queues to share between

  } else {
    swap_chain_create_info.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
    swap_chain_create_info.queueFamilyIndexCount = 0;
    swap_chain_create_info.pQueueFamilyIndices = nullptr;
  }

  // if old swap chain been destroyed and this one replaces it then link old one
  // to quickly hand over responsibilities
  swap_chain_create_info.oldSwapchain = VK_NULL_HANDLE;

  // create swap chain
  VkResult result = vkCreateSwapchainKHR(
      device->getLogicalDevice(), &swap_chain_create_info, nullptr, &swapchain);
  ASSERT_VULKAN(result, "Failed create swapchain!");

  // store for later reference
  swap_chain_image_format = surface_format.format;
  swap_chain_extent = extent;

  // get swapchain images (first count, then values)
  uint32_t swapchain_image_count;
  vkGetSwapchainImagesKHR(device->getLogicalDevice(), swapchain,
                          &swapchain_image_count, nullptr);
  std::vector<VkImage> images(swapchain_image_count);
  vkGetSwapchainImagesKHR(device->getLogicalDevice(), swapchain,
                          &swapchain_image_count, images.data());

  swap_chain_images.clear();

  for (size_t i = 0; i < images.size(); i++) {
    VkImage image = images[static_cast<uint32_t>(i)];
    // store image handle
    Texture swap_chain_image{};
    swap_chain_image.setImage(image);
    swap_chain_image.createImageView(device, swap_chain_image_format,
                                     VK_IMAGE_ASPECT_COLOR_BIT, 1);

    // add to swapchain image list
    swap_chain_images.push_back(swap_chain_image);
  }
}

void VulkanSwapChain::cleanUp() {
  for (Texture& image : swap_chain_images) {
    vkDestroyImageView(device->getLogicalDevice(), image.getImageView(),
                       nullptr);
  }

  vkDestroySwapchainKHR(device->getLogicalDevice(), swapchain, nullptr);
}

VulkanSwapChain::~VulkanSwapChain() {}

VkSurfaceFormatKHR VulkanSwapChain::choose_best_surface_format(
    const std::vector<VkSurfaceFormatKHR>& formats) {
  // best format is subjective, but I go with:
  //  Format:           VK_FORMAT_R8G8B8A8_UNORM (backup-format:
  //  VK_FORMAT_B8G8R8A8_UNORM) color_space:  VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
  //  the condition in if means all formats are available (no restrictions)
  if (formats.size() == 1 && formats[0].format == VK_FORMAT_UNDEFINED) {
    return {VK_FORMAT_R8G8B8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR};
  }

  // if restricted, search  for optimal format
  for (const auto& format : formats) {
    if ((format.format == VK_FORMAT_R8G8B8A8_UNORM ||
         format.format == VK_FORMAT_B8G8R8A8_UNORM) &&
        format.colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
      return format;
    }
  }

  // in case just return first one--- but really shouldn't be the case ....
  return formats[0];
}

VkPresentModeKHR VulkanSwapChain::choose_best_presentation_mode(
    const std::vector<VkPresentModeKHR>& presentation_modes) {
  // look for mailbox presentation mode
  for (const auto& presentation_mode : presentation_modes) {
    if (presentation_mode == VK_PRESENT_MODE_MAILBOX_KHR) {
      return presentation_mode;
    }
  }

  // if can't find, use FIFO as Vulkan spec says it must be present
  return VK_PRESENT_MODE_FIFO_KHR;
}

VkExtent2D VulkanSwapChain::choose_swap_extent(
    const VkSurfaceCapabilitiesKHR& surface_capabilities) {
  // if current extent is at numeric limits, than extent can vary. Otherwise it
  // is size of window
  if (surface_capabilities.currentExtent.width !=
      std::numeric_limits<uint32_t>::max()) {
    return surface_capabilities.currentExtent;

  } else {
    int width, height;
    glfwGetFramebufferSize(window->get_window(), &width, &height);

    // create new extent using window size
    VkExtent2D new_extent{};
    new_extent.width = static_cast<uint32_t>(width);
    new_extent.height = static_cast<uint32_t>(height);

    // surface also defines max and min, so make sure within boundaries bly
    // clamping value
    new_extent.width = std::max(
        surface_capabilities.minImageExtent.width,
        std::min(surface_capabilities.maxImageExtent.width, new_extent.width));
    new_extent.height =
        std::max(surface_capabilities.minImageExtent.height,
                 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),
      x_change(0.0f),
      y_change(0.0f),
      framebuffer_resized(false)

{
  // all keys non-pressed in the beginning
  for (size_t i = 0; i < 1024; i++) {
    keys[i] = 0;
  }

  initialize();
}

// please use this constructor; never the standard
Window::Window(uint32_t window_width, uint32_t window_height)
    :

      window_width(window_width),
      window_height(window_height),
      x_change(0.0f),
      y_change(0.0f),
      framebuffer_resized(false)

{
  // all keys non-pressed in the beginning
  for (size_t i = 0; i < 1024; i++) {
    keys[i] = 0;
  }

  initialize();
}

int Window::initialize() {
  glfwSetErrorCallback(onErrorCallback);
  if (!glfwInit()) {
    printf("GLFW Init failed!");
    glfwTerminate();
    return 1;
  }

  if (!glfwVulkanSupported()) {
    throw std::runtime_error("No Vulkan Supported!");
  }

  // allow it to resize
  glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);

  // retrieve new window
  glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
  main_window =
      glfwCreateWindow(window_width, window_height,
                       "\\__/ Epic graphics from hell \\__/ ", NULL, NULL);

  if (!main_window) {
    printf("GLFW Window creation failed!");
    glfwTerminate();
    return 1;
  }

  // get buffer size information
  glfwGetFramebufferSize(main_window, &window_buffer_width,
                         &window_buffer_height);

  init_callbacks();

  return 0;
}

void Window::cleanUp() {
  glfwDestroyWindow(main_window);
  glfwTerminate();
}

void Window::update_viewport() {
  glfwGetFramebufferSize(main_window, &window_buffer_width,
                         &window_buffer_height);
}

void Window::set_buffer_size(float window_buffer_width,
                             float window_buffer_height) {
  this->window_buffer_width = window_buffer_width;
  this->window_buffer_height = window_buffer_height;
}

float Window::get_x_change() {
  float the_change = x_change;
  x_change = 0.0f;
  return the_change;
}

float Window::get_y_change() {
  float the_change = y_change;
  y_change = 0.0f;
  return the_change;
}

float Window::get_height() { return float(window_height); }

float Window::get_width() { return float(window_width); }

bool Window::framebuffer_size_has_changed() { return framebuffer_resized; }

void Window::init_callbacks() {
  // TODO: remember this section for our later game logic
  // for the space ship to fly around
  glfwSetWindowUserPointer(main_window, this);
  glfwSetKeyCallback(main_window, &key_callback);
  glfwSetMouseButtonCallback(main_window, &mouse_button_callback);
  glfwSetFramebufferSizeCallback(main_window, &framebuffer_size_callback);
}

void Window::framebuffer_size_callback(GLFWwindow* window, int width,
                                       int height) {
  auto app = reinterpret_cast<Window*>(glfwGetWindowUserPointer(window));
  app->framebuffer_resized = true;
  app->window_width = width;
  app->window_height = height;
}

void Window::reset_framebuffer_has_changed() {
  this->framebuffer_resized = false;
}

void Window::key_callback(GLFWwindow* window, int key, int code, int action,
                          int mode) {
  Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));

  if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) {
    glfwSetWindowShouldClose(window, VK_TRUE);
  }

  if (key >= 0 && key < 1024) {
    if (action == GLFW_PRESS) {
      the_window->keys[key] = true;

    } else if (action == GLFW_RELEASE) {
      the_window->keys[key] = false;
    }
  }
}

void Window::mouse_callback(GLFWwindow* window, double x_pos, double y_pos) {
  Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));

  // need to handle first occurance of a mouse moving event
  if (the_window->mouse_first_moved) {
    the_window->last_x = static_cast<float>(x_pos);
    the_window->last_y = static_cast<float>(y_pos);
    the_window->mouse_first_moved = false;
  }

  the_window->x_change = static_cast<float>((x_pos - the_window->last_x));
  // take care of correct substraction :)
  the_window->y_change = static_cast<float>((the_window->last_y - y_pos));

  // update params
  the_window->last_x = static_cast<float>(x_pos);
  the_window->last_y = static_cast<float>(y_pos);
}

void Window::mouse_button_callback(GLFWwindow* window, int button, int action,
                                   int mods) {
  if (ImGui::GetCurrentContext() != nullptr &&
      ImGui::GetIO().WantCaptureMouse) {
    ImGuiIO& io = ImGui::GetIO();
    io.AddMouseButtonEvent(button, action);
    return;
  }

  Window* the_window = static_cast<Window*>(glfwGetWindowUserPointer(window));

  if ((action == GLFW_PRESS) && (button == GLFW_MOUSE_BUTTON_RIGHT)) {
    glfwSetCursorPosCallback(window, mouse_callback);
  } else {
    the_window->mouse_first_moved = true;
    glfwSetCursorPosCallback(window, NULL);
  }
}

Window::~Window() {}
```

Index