ДОДАТОК А

А.1 Лістинг програмної реалізації файлу Render.cpp

```
#include "RenderJob.h"
#include "GuiInterface.h"
#include "vulkan_impl/VulkanCommandRecorder.h"
#include "vulkan_impl/VulkanDevice.h"
#include "vulkan_impl/VulkanPipeline.h"
#include "vulkan_impl/VulkanRenderImpl.h"
#include "vulkan_impl/VulkanSwapChain.h"
#include <cassert>

APITest::RenderJob::RenderJob(APITest::VulkanRenderImpl
*parent,
                              RenderPassRef desc)
    : parent_(parent) {
  compileRenderPass(desc);
}

void       APITest::RenderJob::compileRenderPass(RenderPassRef
desc) {
  auto pass = desc.lock();
  for (auto const &dep : pass->dependencies)
    compileRenderPass(dep);

  auto *vkPass = dynamic_cast<VulkanRenderPass *>(pass.get());

  vkPass->compile();

  auto &unit = renderSequence.emplace_back(desc);

                  if        (auto        *onScreenPass        =
dynamic_cast<VulkanOnscreenRenderPass *>(vkPass)) {
    auto *swapChain = parent_->getSwapChain();
    int imageCount = swapChain->buffers.size();
    for (int i = 0; i < imageCount; i++) {
      unit.frameBuffers.push_back(
                   onScreenPass->allocateFrameBuffer(swapChain-
>buffers.at(i).view));
    }
  } else {
       throw  std::runtime_error("[Vulkan][ERROR]  Render  Pass
graph compilation "
                            "failed: offscreen passes aren't
supported yet.");
  }
}
```

```
    void APITest::RenderJob::resetFrameBuffers() {
      for (auto &unit : renderSequence) {
        auto pass = unit.pass.lock();
        if (auto *onScreenPass =
                                dynamic_cast<VulkanOnscreenRenderPass
*>(pass.get())) {
          for (auto buffer : unit.frameBuffers)
                vkDestroyFramebuffer(parent_->getVulkanDevice()-
>get(), buffer,
                                     nullptr);
          unit.frameBuffers.clear();
          auto *swapChain = parent_->getSwapChain();
          int imageCount = swapChain->buffers.size();
          onScreenPass->changeExtents(swapChain->surfaceWidth,
                                      swapChain->surfaceHeight);
          for (int i = 0; i < imageCount; i++) {
            unit.frameBuffers.push_back(
                    onScreenPass->allocateFrameBuffer(swapChain-
>buffers.at(i).view));
          }
        }
      }
    }

    void APITest::RenderJob::compile(int swapChainImage,
                                                        VkCommandBuffer
commandBuffer) {

      auto &statRecorder = parent_->recorder();

      statRecorder.push("Command record");

      VulkanCommandRecorder commandRecorder;

      commandRecorder.bufferToRecord = commandBuffer;

      for (auto &unit : renderSequence) {
        auto pass = unit.pass.lock();
        statRecorder.stamp(
            "Render Pass " +
                    std::to_string(reinterpret_cast<unsigned  long
long>(pass.get())));
                auto   *vkPass   =   dynamic_cast<VulkanRenderPass
*>(pass.get());
        auto currentBuffer = unit.frameBuffers.size() == 1
                                ? unit.frameBuffers.front()
                                                                    :
unit.frameBuffers.at(swapChainImage);
        vkPass->begin(commandBuffer, currentBuffer);

        commandRecorder.currentPass = vkPass;
```

```
        vkPass->commands(&commandRecorder);

        if (dynamic_cast<OnscreenRenderPass *>(vkPass)) {
          auto *gui = parent_->getGuiInterface();
          if (gui)
            gui->draw(&commandRecorder);
        }

        vkPass->end(commandBuffer);
      }

      statRecorder.pop();
    }

    APITest::RenderJob::~RenderJob() {
      for (auto &unit : renderSequence)
        for (auto buffer : unit.frameBuffers)
           vkDestroyFramebuffer(parent_->getVulkanDevice()->get(),
buffer, nullptr);
    }
```

А.2 Лістинг програмної реалізації файлу VulkanCommandRecorder.cpp

```
    #include "VulkanCommandRecorder.h"
    #include "vulkan_impl/VulkanDescriptorManager.h"
    #include "vulkan_impl/VulkanMemoryManager.h"
    #include "vulkan_impl/VulkanPipeline.h"
    #include <cassert>
    #include <stdexcept>

    void
APITest::VulkanCommandRecorder::bindPipeline(APITest::Pipeline
*pipeline) {
      currentPipeline = dynamic_cast<VulkanPipeline *>(pipeline);
      if (auto *vulkanPipeline =
                                dynamic_cast<VulkanGraphicsPipeline
*>(currentPipeline)) {
        vkCmdBindPipeline(bufferToRecord,
                          VK_PIPELINE_BIND_POINT_GRAPHICS /* TODO
*/,
                          vulkanPipeline->get(currentPass));
        pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
      } else {
        throw std::runtime_error(
              "[Vulkan][ERROR] bindPipeline() - invalid pipeline
object.");
      }
    }

    void             APITest::VulkanCommandRecorder::draw(uint32_t
vertexCount,
```

```
                                                                uint32_t
firstIndex) {
       vkCmdDraw(bufferToRecord, vertexCount, 1, firstIndex, 0);
     }

     static                                                  VkIndexType
convertIndexType(APITest::IndexBuffer::Type type) {
       switch (type) {
       case APITest::IndexBuffer::Type::INDEX_TYPE_UINT_32:
         return VK_INDEX_TYPE_UINT32;
       case APITest::IndexBuffer::Type::INDEX_TYPE_UINT_16:
         return VK_INDEX_TYPE_UINT16;
       case APITest::IndexBuffer::Type::INDEX_TYPE_UINT_8:
         return VK_INDEX_TYPE_UINT8_EXT;
       }

        throw  std::runtime_error("[VULKAN][ERROR]:  invalid  index
type");
     }

     void APITest::VulkanCommandRecorder::bindVertexBuffer(
         APITest::VertexBuffer *buffer, uint32_t binding) {
       auto *vkBuffer = dynamic_cast<VulkanVertexBuffer *>(buffer);
       if (!vkBuffer)
         throw std::runtime_error(
                   "[VULKAN][ERROR]:  invalid  vertex  buffer  object
used.");
       VkDeviceSize offset = 0;
       auto buf = vkBuffer->buffer();
        vkCmdBindVertexBuffers(bufferToRecord,  binding,  1,  &buf,
&offset);
     }

     void APITest::VulkanCommandRecorder::bindDescriptorSet(
         APITest::UniformDescriptorSet *set) {
          if   (auto   *castSet   =   dynamic_cast<VulkanDescriptorSet
*>(set)) {
         auto vkSet = castSet->get();
         vkCmdBindDescriptorSets(bufferToRecord, pipelineBindPoint,
                                 currentPipeline->getLayout(), 0,
1, &vkSet, 0,
                                 nullptr);
       } else {
         throw std::runtime_error(
                   "[Vulkan][ERROR]  bindDescriptorSet()  -  invalid
descriptor set object.");
       }
     }

     void      APITest::VulkanCommandRecorder::drawIndexed(uint32_t
indexCount,
```

```
                                                                uint32_t
firstIndex,
                                                                uint32_t
vertexOffset) {
        vkCmdDrawIndexed(bufferToRecord, indexCount, 1, firstIndex,
vertexOffset, 0);
    }

    void APITest::VulkanCommandRecorder::bindIndexBuffer(
        APITest::IndexBuffer *buffer) {
      auto *indexBuf = dynamic_cast<VulkanIndexBuffer *>(buffer);

      VkDeviceSize offset = 0;

      auto buf = indexBuf->buffer();

      vkCmdBindIndexBuffer(bufferToRecord, buf, offset,
                           convertIndexType(indexBuf->type));
    }
```

A.3 Лістинг програмної реалізації файлу VulkanDescriptionManager.cpp

```
    #include "VulkanDescriptorManager.h"
    #include "VulkanDevice.h"
    #include "VulkanMemoryManager.h"
    #include "VulkanRenderImpl.h"
    #include "VulkanSampler.h"
    #include "util/VulkanInitializers.h"
    #include "util/macro.h"

    #include <cassert>

    namespace APITest {

    static VkDescriptorType
    convertInterfaceDescriptorType(APITest::UniformDescriptor
const &desc) {
                                                                if
(std::holds_alternative<APITest::UniformBufferRef>(desc.descriptor
))
        return VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
                                                                if
(std::holds_alternative<APITest::CombinedImageSampler>(desc.descri
ptor))
        return VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;

           throw   std::runtime_error("[VULKAN][ERROR]:   invalid
descriptor type.");
    }

    static VkDescriptorType
```

```
    convertInterfaceDescriptorType(APITest::DescriptorLayout::Typ
e type) {
      switch (type) {
      case APITest::DescriptorLayout::Type::UNIFORM_BUFFER:
        return VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
                                                              case
APITest::DescriptorLayout::Type::COMBINED_IMAGE_SAMPLER:
        return VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
      }
           throw   std::runtime_error("[VULKAN][ERROR]:   invalid
descriptor type.");
    }

    static                                        VkShaderStageFlags
convertStageFlags(APITest::ShaderStage stage) {
      switch (stage) {
      case ShaderStage::VERTEX:
        return VK_SHADER_STAGE_VERTEX_BIT;
      case ShaderStage::FRAGMENT:
        return VK_SHADER_STAGE_FRAGMENT_BIT;
      }
       throw  std::runtime_error("[VULKAN][ERROR]:  invalid  shader
stage.");
    }

    VulkanDescriptorSet::VulkanDescriptorSet(VulkanDescriptorPool
*parent,
                                             VkDescriptorSetLayou
t layout,
                                                UniformDescriptor
*descriptor,
                                            int count) {

      VkDescriptorSetAllocateInfo allocateInfo =
            initializers::descriptorSetAllocateInfo(parent->pool,
&layout, 1);

      VK_CHECK_RESULT(
                          vkAllocateDescriptorSets(parent->parent_-
>getVulkanDevice()->get(),
                                                   &allocateInfo,
&descriptorSet_))

      std::vector<VkWriteDescriptorSet> writes;
             std::vector<std::unique_ptr<VkDescriptorBufferInfo>>
bufferInfos;
              std::vector<std::unique_ptr<VkDescriptorImageInfo>>
imageInfos;
      for (int i = 0; i < count; i++) {
        auto &desc = descriptor[i];
```

```
                                                                    if
(std::holds_alternative<APITest::UniformBufferRef>(desc.descriptor
)) {
                                        auto     uniformBuffer     =
std::get<APITest::UniformBufferRef>(desc.descriptor);
            auto &bufferInfo =
                 bufferInfos.emplace_back(std::make_unique<VkDescrip
torBufferInfo>());
            auto *bufferImpl =
                                      dynamic_cast<VulkanUniformBuffer
*>(uniformBuffer.get());
            if (!bufferImpl)
                 throw std::runtime_error("[VULKAN][ERROR]: Invalid
UniformBuffer "
                                          "object passed to descriptor
set");
            bufferInfo->buffer = bufferImpl->buffer();
            bufferInfo->offset = 0;
            bufferInfo->range = bufferImpl->size();
                                   VkWriteDescriptorSet    write    =
initializers::writeDescriptorSet(
                  descriptorSet_, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
desc.binding,
                bufferInfo.get());
            writes.push_back(write);
                                        }         else         if
(std::holds_alternative<APITest::CombinedImageSampler>(
                        desc.descriptor)) {
            auto combinedImageSampler =
                 std::get<APITest::CombinedImageSampler>(desc.descri
ptor);
            auto &imageInfo =
                 imageInfos.emplace_back(std::make_unique<VkDescript
orImageInfo>());
            imageInfo->sampler =
                                         dynamic_cast<VulkanSampler
*>(combinedImageSampler.first.get())
                    ->get();
                                        imageInfo->imageView      =
dynamic_cast<VulkanImageInterface *>(
                                       combinedImageSampler.second.
get())
                                      ->getDefaultView();
             // if we sample from image, assume we did our best to
convert it to this
            // layout to the point
                                       imageInfo->imageLayout     =
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
                                   VkWriteDescriptorSet    write    =
initializers::writeDescriptorSet(
                                                          descriptorSet_,
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
```

```
                desc.binding, imageInfo.get());
            writes.push_back(write);
          }
        }

          vkUpdateDescriptorSets(parent->parent_->getVulkanDevice()-
>get(),
                                 writes.size(), writes.data(), 0,
nullptr);
      }

      VulkanDescriptorPool::VulkanDescriptorPool(
                            VulkanRenderImpl     *parent,     const
std::vector<VkDescriptorPoolSize> &sizes,
          int maxSets)
          : parent_(parent), maxSets_(maxSets) {
        descriptors.resize(sizes.size());
        std::transform(
            sizes.begin(), sizes.end(), descriptors.begin(),
            [](VkDescriptorPoolSize size) {
               return std::pair<VkDescriptorType, std::pair<size_t,
size_t>>{
                                     size.type,   std::pair<size_t,
size_t>{size.descriptorCount, 0}};
            });
        VkDescriptorPoolCreateInfo createInfo =
            initializers::descriptorPoolCreateInfo(sizes, maxSets_);
                  VK_CHECK_RESULT(vkCreateDescriptorPool(parent_-
>getVulkanDevice()->get(),
                                                         &createInfo,
nullptr, &pool))
      }

      VulkanDescriptorPool::~VulkanDescriptorPool() {
          vkDestroyDescriptorPool(parent_->getVulkanDevice()->get(),
pool, nullptr);
      }

      UniformDescriptorSetRef
VulkanDescriptorPool::allocateDescriptorSet(
                VkDescriptorSetLayout   layout,   UniformDescriptor
*descriptor, int count) {
        std::vector<VkDescriptorPoolSize> totalSizes;
        for (int i = 0; i < count; i++) {
          auto type = convertInterfaceDescriptorType(descriptor[i]);
          auto found = std::find_if(
              totalSizes.begin(), totalSizes.end(),
                [type](VkDescriptorPoolSize size) { return type ==
size.type; });
          if (found != totalSizes.end())
            found->descriptorCount++;
          else
```

```
            totalSizes.emplace_back(VkDescriptorPoolSize{type, 1});
        }

        for (auto &descTypeCount : descriptors) {
                auto found = std::find_if(totalSizes.begin(),
totalSizes.end(),
                                          [descTypeCount]
(VkDescriptorPoolSize size) {
                                            return descTypeCount.first ==
size.type;
                                          });
          if (found == totalSizes.end())
            continue;

          if (found->descriptorCount + descTypeCount.second.second >
              descTypeCount.second.first)
            return nullptr;
          descTypeCount.second.second += found->descriptorCount;
        }

        allocatedSets.emplace_back(UniformDescriptorSetRef(
                new VulkanDescriptorSet(this, layout, descriptor,
count)));
        return allocatedSets.back();
      }

      VulkanDescriptorSetLayout::VulkanDescriptorSetLayout(
                        VulkanDescriptorManager *parent, const
std::vector<DescriptorLayout> &desc)
          : parent_(parent) {
        std::vector<VkDescriptorSetLayoutBinding> bindings;
        for (auto &layout : desc) {
          VkDescriptorSetLayoutBinding binding{};
          binding.binding = layout.binding;
                                    binding.descriptorType =
convertInterfaceDescriptorType(layout.type);
          binding.stageFlags = convertStageFlags(layout.stage);
          binding.descriptorCount = 1;
          bindings.emplace_back(binding);
        }
        VkDescriptorSetLayoutCreateInfo createInfo =
            initializers::descriptorSetLayoutCreateInfo(bindings);

        VK_CHECK_RESULT(
                        vkCreateDescriptorSetLayout(parent_->parent_-
>getVulkanDevice()->get(),
                                                    &createInfo, nullptr,
&setLayout_))
      }

      VulkanDescriptorSetLayout::~VulkanDescriptorSetLayout() {
```

```
                    vkDestroyDescriptorSetLayout(parent_->parent_-
>getVulkanDevice()->get(),
                                    setLayout_, nullptr);
    }

    UniformDescriptorSetRef
    VulkanDescriptorSetLayout::allocateNewSet(UniformDescriptor
*descriptors,
                                              int count) {
             return    parent_->allocateDescriptorSet(setLayout_,
descriptors, count);
    }
    } // namespace APITest

    APITest::UniformDescriptorSetRef
    APITest::VulkanDescriptorManager::allocateDescriptorSet(
         VkDescriptorSetLayout layout, APITest::UniformDescriptor
*descriptor,
        int count) {
      UniformDescriptorSetRef ret;

      for (auto &pool : pools_) {
           ret = pool->allocateDescriptorSet(layout, descriptor,
count);
        if (ret)
          return std::move(ret);
      }
      std::vector<VkDescriptorPoolSize> sizes;
      for (int i = 0; i < count; i++) {
        auto type = convertInterfaceDescriptorType(descriptor[i]);
        auto size = std::find_if(sizes.begin(), sizes.end(),
                                 [type](VkDescriptorPoolSize
const &poolSize) {
                                   return poolSize.type == type;
                                 });
        if (size != sizes.end())
          size->descriptorCount++;
        else
          sizes.emplace_back(VkDescriptorPoolSize{type, 1});
      }

       // for now we will try to create new pool for 100 such
descriptor sets if
      // count is low(less than 10) and for 1000 if above

      int allocateForSets = count > 10 ? 100 : 1000;

      for (auto &size : sizes)
        size.descriptorCount *= allocateForSets;

      pools_.emplace_back(
```

```
            std::make_unique<VulkanDescriptorPool>(parent_, sizes,
allocateForSets));

           ret    =    pools_.back()->allocateDescriptorSet(layout,
descriptor, count);
      if (!ret)
        throw std::runtime_error(
            "[VULKAN][ERROR]: cannot allocate descriptor pool.");
      return ret;
    }
```

А.4 Лістинг програмної реалізації файлу VulkanDevice.cpp

```
    #include "VulkanDevice.h"
    #include "VulkanRenderImpl.h"
    #include <cassert>
    #include <iostream>

    APITest::VulkanDevice::VulkanDevice(const
APITest::VulkanRenderImpl *parent,
                                        bool useSwapChain)
        : parent_(parent) {
      uint32_t deviceCount = 0;
               vkEnumeratePhysicalDevices(parent_->getInstance(),
&deviceCount, nullptr);
      if (deviceCount == 0)
        throw std::runtime_error("There are no devices supporting
Vulkan.");

      std::vector<VkPhysicalDevice> availableDevices(deviceCount);
               vkEnumeratePhysicalDevices(parent_->getInstance(),
&deviceCount,
                                  availableDevices.data());

      physicalDevice_ = availableDevices[0];

      queryPhysicalDeviceInfo();

      for (int i = 0; i < availableDevices.size(); i++) {
        try {
          createLogicalDevice(useSwapChain);
        } catch (std::runtime_error &e) {
          if (i == availableDevices.size() - 1)
            throw e;

          continue;
        }
      }
    }
```

```
    void APITest::VulkanDevice::queryPhysicalDeviceInfo() {
      // Store Properties features, limits and properties of the
physical device for
       // later use Device properties also contain limits and
sparse properties
      vkGetPhysicalDeviceProperties(physicalDevice_, &properties);
       // Features should be checked by the examples before using
them
      vkGetPhysicalDeviceFeatures(physicalDevice_, &features);
       // Memory properties are used regularly for creating all
kinds of buffers
              vkGetPhysicalDeviceMemoryProperties(physicalDevice_,
&memoryProperties);
       // Queue family properties, used for setting up requested
queues upon device
      // creation
      uint32_t queueFamilyCount;
         vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice_,
&queueFamilyCount,
                                                  nullptr);
      assert(queueFamilyCount > 0);
      queueFamilyProperties.resize(queueFamilyCount);
         vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice_,
&queueFamilyCount,
                                                   queueFamilyPropert
ies.data());

      // Get list of supported extensions
      uint32_t extCount = 0;
             vkEnumerateDeviceExtensionProperties(physicalDevice_,
nullptr, &extCount,
                                                  nullptr);
      if (extCount > 0) {
        std::vector<VkExtensionProperties> extensions(extCount);
         if (vkEnumerateDeviceExtensionProperties(physicalDevice_,
nullptr,
                                                              &extCount,
&extensions.front()) ==
             VK_SUCCESS) {
          for (auto ext : extensions) {
            supportedExtensions.emplace_back(ext.extensionName);
          }
        }
      }

      std::cout << "Vulkan picked following GPU for a test: "
                << properties.deviceName << std::endl;
    }

    bool           APITest::VulkanDevice::createLogicalDevice(bool
useSwapChain) {
```

```
          auto requestedQueueTypes = VK_QUEUE_COMPUTE_BIT |
VK_QUEUE_GRAPHICS_BIT;
       // Desired queues need to be requested upon logical device
creation
       // Due to differing queue family configurations of Vulkan
implementations this
       // can be a bit tricky, especially if the application
requests different queue
       // types

       std::vector<VkDeviceQueueCreateInfo> queueCreateInfos{};

       // Get queue family indices for the requested queue family
types
       // Note that the indices may overlap depending on the
implementation

       const float defaultQueuePriority(0.0f);

       // Graphics queue
       if (requestedQueueTypes & VK_QUEUE_GRAPHICS_BIT) {
                                  queueFamilyIndices.graphics =
getQueueFamilyIndex(VK_QUEUE_GRAPHICS_BIT);
         VkDeviceQueueCreateInfo queueInfo{};
                                           queueInfo.sType =
VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
         queueInfo.queueFamilyIndex = queueFamilyIndices.graphics;
         queueInfo.queueCount = 1;
         queueInfo.pQueuePriorities = &defaultQueuePriority;
         queueCreateInfos.push_back(queueInfo);
       } else {
         queueFamilyIndices.graphics = 0;
       }

       // Dedicated compute queue
       if (requestedQueueTypes & VK_QUEUE_COMPUTE_BIT) {
                                   queueFamilyIndices.compute =
getQueueFamilyIndex(VK_QUEUE_COMPUTE_BIT);
                           if (queueFamilyIndices.compute !=
queueFamilyIndices.graphics) {
               // If compute family index differs, we need an
additional queue create
           // info for the compute queue
           VkDeviceQueueCreateInfo queueInfo{};
                                             queueInfo.sType =
VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
           queueInfo.queueFamilyIndex = queueFamilyIndices.compute;
           queueInfo.queueCount = 1;
           queueInfo.pQueuePriorities = &defaultQueuePriority;
           queueCreateInfos.push_back(queueInfo);
         }
       } else {
```

```
        // Else we use the same queue
        queueFamilyIndices.compute = queueFamilyIndices.graphics;
      }

      // Dedicated transfer queue
      if (requestedQueueTypes & VK_QUEUE_TRANSFER_BIT) {
                                queueFamilyIndices.transfer       =
getQueueFamilyIndex(VK_QUEUE_TRANSFER_BIT);
                        if     ((queueFamilyIndices.transfer     !=
queueFamilyIndices.graphics) &&
                                  (queueFamilyIndices.transfer    !=
queueFamilyIndices.compute)) {
              // If compute family index differs, we need an
additional queue create
          // info for the compute queue
          VkDeviceQueueCreateInfo queueInfo{};
                                              queueInfo.sType       =
VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
                                   queueInfo.queueFamilyIndex     =
queueFamilyIndices.transfer;
          queueInfo.queueCount = 1;
          queueInfo.pQueuePriorities = &defaultQueuePriority;
          queueCreateInfos.push_back(queueInfo);
        }
      } else {
        // Else we use the same queue
        queueFamilyIndices.transfer = queueFamilyIndices.graphics;
      }

      // Create the logical device representation
      std::vector<const char *> deviceExtensions{};
      if (useSwapChain) {
        // If the device will be used for presenting to a display
via a swapchain we
        // need to request the swapchain extension
         deviceExtensions.push_back(VK_KHR_SWAPCHAIN_EXTENSION_NAM
E);
      }

      VkDeviceCreateInfo deviceCreateInfo = {};
                               deviceCreateInfo.sType             =
VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
      deviceCreateInfo.queueCreateInfoCount =
          static_cast<uint32_t>(queueCreateInfos.size());
                      deviceCreateInfo.pQueueCreateInfos         =
queueCreateInfos.data();
      deviceCreateInfo.pEnabledFeatures = &enabledFeatures;

       // Enable the debug marker extension if it is present
(likely meaning a
      // debugging tool is present)
```

```
        if (extensionSupported(VK_EXT_DEBUG_MARKER_EXTENSION_NAME))
{
          deviceExtensions.push_back(VK_EXT_DEBUG_MARKER_EXTENSION_
NAME);
         enableDebugMarkers = true;
       }

       if (!deviceExtensions.empty()) {
         for (const char *enabledExtension : deviceExtensions) {
           if (!extensionSupported(enabledExtension)) {
             throw std::runtime_error(
                       "[VULKAN][ERROR] device has no support for
enabled extension " +
                   std::string(enabledExtension));
           }
         }

                        deviceCreateInfo.enabledExtensionCount     =
(uint32_t)deviceExtensions.size();
                      deviceCreateInfo.ppEnabledExtensionNames     =
deviceExtensions.data();
       }

           VkResult   result   =   vkCreateDevice(physicalDevice_,
&deviceCreateInfo, nullptr,
                                             &logicalDevice_);
       if (result != VK_SUCCESS) {
             throw  std::runtime_error("[VULKAN][ERROR]  failed  to
create device");
       }

       return result;
     }

     uint32_t
     APITest::VulkanDevice::getQueueFamilyIndex(VkQueueFlagBits
queueFlags) const {
       // Dedicated queue for compute
        // Try to find a queue family index that supports compute
but not graphics
       if (queueFlags & VK_QUEUE_COMPUTE_BIT) {
         for (uint32_t i = 0;
                                                          i      <
static_cast<uint32_t>(queueFamilyProperties.size()); i++) {
            if ((queueFamilyProperties[i].queueFlags & queueFlags)
&&
                           ((queueFamilyProperties[i].queueFlags  &
VK_QUEUE_GRAPHICS_BIT) ==
                0)) {
             return i;
           }
         }
```

```
      }

      // Dedicated queue for transfer
      // Try to find a queue family index that supports transfer
but not graphics
      // and compute
      if (queueFlags & VK_QUEUE_TRANSFER_BIT) {
        for (uint32_t i = 0;
                                                          i      <
static_cast<uint32_t>(queueFamilyProperties.size()); i++) {
          if ((queueFamilyProperties[i].queueFlags & queueFlags)
&&
                        ((queueFamilyProperties[i].queueFlags  &
VK_QUEUE_GRAPHICS_BIT) ==
              0) &&
                        ((queueFamilyProperties[i].queueFlags  &
VK_QUEUE_COMPUTE_BIT) == 0)) {
            return i;
          }
        }
      }

      // For other queue types or if no separate compute queue is
present, return
      // the first one to support the requested flags
              for     (uint32_t     i     =     0;     i     <
static_cast<uint32_t>(queueFamilyProperties.size());
          i++) {
        if (queueFamilyProperties[i].queueFlags & queueFlags) {
          return i;
        }
      }

       throw std::runtime_error("Could not find a matching queue
family index");
    }

    bool     APITest::VulkanDevice::extensionSupported(std::string
const &extension) {
              return      (std::find(supportedExtensions.begin(),
supportedExtensions.end(),
                      extension) != supportedExtensions.end());
    }

    VkFormat APITest::VulkanDevice::getSupportedDepthFormat(
        bool checkSamplingSupport) const {
      // All depth formats may be optional, so we need to find a
suitable depth
      // format to use
      std::vector<VkFormat> depthFormats = {
          VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D32_SFLOAT,
```

```
                                        VK_FORMAT_D24_UNORM_S8_UINT,
VK_FORMAT_D16_UNORM_S8_UINT,
          VK_FORMAT_D16_UNORM};
       for (auto &format : depthFormats) {
         VkFormatProperties formatProperties;
               vkGetPhysicalDeviceFormatProperties(physicalDevice_,
format,
                                                   &formatProperties);
           // Format must support depth stencil attachment for
optimal tiling
         if (formatProperties.optimalTilingFeatures &
             VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT) {
           if (checkSamplingSupport) {
             if (!(formatProperties.optimalTilingFeatures &
                   VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT)) {
               continue;
             }
           }
           return format;
         }
       }
        throw std::runtime_error("Could not find a matching depth
format");
     }

     APITest::VulkanDevice::~VulkanDevice() {
       if (logicalDevice_) {
         vkDestroyDevice(logicalDevice_, nullptr);
       }
     }

     void                          APITest::VulkanDevice::waitIdle()
{ vkDeviceWaitIdle(logicalDevice_); }
```

A.5 Лістинг програмної реалізації файлу VulkanMemoryManager.cpp

```
     #include "VulkanMemoryManager.h"
     #include "VulkanDevice.h"
     #include "VulkanQueueManager.h"
     #include "VulkanRenderImpl.h"
     #include "util/VulkanInitializers.h"
     #include <cassert>
     #include <cstring>
     #include <vulkan_impl/util/macro.h>

     APITest::VulkanBuffer::~VulkanBuffer() {
       if (memoryManager != VK_NULL_HANDLE) {
             vmaDestroyBuffer(memoryManager->allocator(),  buffer_,
allocation);
       }
     }
```

```
    namespace APITest {
    static                                          VkImageUsageFlags
convertInterfaceImageUsageFlags(Image::Usage usage) {
      VkImageUsageFlags ret = 0;

      if (usage & Image::USAGE_COPY_TO)
        ret |= VK_IMAGE_USAGE_TRANSFER_DST_BIT;
      if (usage & Image::USAGE_COPY_FROM)
        ret |= VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
      if (usage & Image::USAGE_SAMPLED)
        ret |= VK_IMAGE_USAGE_SAMPLED_BIT;
      if (usage & Image::USAGE_INPUT_ATTACHMENT)
        ret |= VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT;
      if (usage & Image::USAGE_DEPTH_STENCIL_ATTACHMENT)
        ret |= VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
      if (usage & Image::USAGE_COLOR_ATTACHMENT)
        ret |= VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
      return ret;
    }
    static VmaAllocationCreateInfo chooseMemoryOptions(MemoryType
type) {
      VmaAllocationCreateInfo allocInfo = {};
        allocInfo.usage  =  type  ==  MemoryType::GPU_PRIVATE  ?
VMA_MEMORY_USAGE_GPU_ONLY
                          : type == MemoryType::HOST_VISIBLE
                              ? VMA_MEMORY_USAGE_CPU_TO_GPU
                              : VMA_MEMORY_USAGE_CPU_ONLY;
      allocInfo.preferredFlags = type == MemoryType::HOST_COHERENT
                                                                  ?
(VK_MEMORY_PROPERTY_HOST_COHERENT_BIT |
                                          VK_MEMORY_PROPERTY_HOST_C
ACHED_BIT)
                                       : 0;
      allocInfo.requiredFlags =
                   type  ==  MemoryType::HOST_VISIBLE  ||  type  ==
MemoryType::HOST_COHERENT
               ? VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
               : VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT;
      allocInfo.flags =
                   type  ==  MemoryType::HOST_VISIBLE  ||  type  ==
MemoryType::HOST_COHERENT
               ? VMA_ALLOCATION_CREATE_MAPPED_BIT
               : 0;

      return allocInfo;
    }

    VkImageCreateInfo
    compileDepthStencilImageCreateInfo(VulkanDepthBufferCI
createInfo) {
      VkImageCreateInfo ret = initializers::imageCreateInfo();
      ret.format = createInfo.depthFormat;
```

```
      ret.imageType = VK_IMAGE_TYPE_2D;
      ret.usage =
                    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT   |
createInfo.additionalUsage;
      ret.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
      ret.samples = VK_SAMPLE_COUNT_1_BIT;
      ret.mipLevels = 1;
      ret.arrayLayers = 1;
          ret.tiling   =   createInfo.additionalUsage   ==   0   ?
VK_IMAGE_TILING_OPTIMAL
                                                                  :
VK_IMAGE_TILING_LINEAR;
                ret.extent      =      {createInfo.extents.width,
createInfo.extents.height, 1};
      ret.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

      return ret;
    }
    VulkanDepthBuffer::VulkanDepthBuffer(const
VulkanMemoryManager *alloc,
                                         VulkanDepthBufferCI
createInfo)
        : VulkanImage(
              alloc, MemoryType::GPU_PRIVATE,
              static_cast<VkBufferUsageFlagBits>(
                  VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT |
                   convertInterfaceImageUsageFlags(createInfo.addi
tionalUsage)),
              compileDepthStencilImageCreateInfo(createInfo)) {
                    VkImageViewCreateInfo       viewInfo       =
initializers::imageViewCreateInfo();
      viewInfo.format = createInfo.depthFormat;
      viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
      viewInfo.image = image();
            viewInfo.components    =    {VK_COMPONENT_SWIZZLE_R,
VK_COMPONENT_SWIZZLE_G,
                                          VK_COMPONENT_SWIZZLE_B,
VK_COMPONENT_SWIZZLE_A};
       viewInfo.subresourceRange = {VK_IMAGE_ASPECT_DEPTH_BIT, 0,
1, 0, 1};

      VK_CHECK_RESULT(
           vkCreateImageView(alloc->device(), &viewInfo, nullptr,
&depthView_))
    }

    VulkanDepthBuffer::~VulkanDepthBuffer() {
      if (depthView_ != VK_NULL_HANDLE) {
          vkDestroyImageView(memoryManager->device(), depthView_,
nullptr);
      }
    }
```

```
    VkImageView VulkanDepthBuffer::getDepthView() const { return
depthView_; }

    Image::Type VulkanImageInterface::getType() const {
      switch (imageInfo_.imageType) {
      case VK_IMAGE_TYPE_1D:
        return Image::Type::TEXTURE_1D;
      case VK_IMAGE_TYPE_2D:
        return Image::Type::TEXTURE_2D;
      case VK_IMAGE_TYPE_3D:
        return Image::Type::TEXTURE_3D;
      default:
        return Image::Type::INVALID;
      }
    }

    Image::Format VulkanImageInterface::getFormat() const {
      switch (imageInfo_.format) {
      case VK_FORMAT_R8G8B8A8_SRGB:
        return Image::Format::RGBA8;
      case VK_FORMAT_D24_UNORM_S8_UINT:
        return Image::Format::D24S8;
      case VK_FORMAT_B10G11R11_UFLOAT_PACK32:
        return Image::Format::R11G11B10;
      default:
        return Image::Format::INVALID;
      }
    }

    Image::Extents VulkanImageInterface::getImageExtents() const {
      return {imageInfo_.extent.width, imageInfo_.extent.height,
              imageInfo_.extent.depth};
    }

    uint32_t VulkanImageInterface::arrayLayers() const {
      return imageInfo_.arrayLayers;
    }

    Image::Usage VulkanImageInterface::getUsage() const {
      Image::Usage ret;
      if (imageInfo_.usage & VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT)
        ret |= Image::USAGE_COLOR_ATTACHMENT;
                          if           (imageInfo_.usage           &
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT)
        ret |= Image::USAGE_DEPTH_STENCIL_ATTACHMENT;
      if (imageInfo_.usage & VK_IMAGE_USAGE_SAMPLED_BIT)
        ret |= Image::USAGE_SAMPLED;
      if (imageInfo_.usage & VK_IMAGE_USAGE_TRANSFER_DST_BIT)
        ret |= Image::USAGE_COPY_TO;
      if (imageInfo_.usage & VK_IMAGE_USAGE_TRANSFER_SRC_BIT)
        ret |= Image::USAGE_COPY_FROM;
```

```
    if (imageInfo_.usage & VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT)
      ret |= Image::USAGE_INPUT_ATTACHMENT;

    return ret;
  }

  void VulkanImageInterface::load(const void *data) {

    if (data == nullptr)
      throw std::runtime_error(
          "[VULKAN][ERROR]: passed nullptr to image->load().");

        size_t imageSize = imageInfo_.extent.width *
imageInfo_.extent.height * 4;

    VulkanStagingBuffer buffer{memoryManager, imageSize};
    buffer.push(data, imageSize, 0);

                              memoryManager->transitLayout(this,
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);

    memoryManager->copy(&buffer, this);

                              memoryManager->transitLayout(this,
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
    currentLayout_ = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
  }

  static VkImageAspectFlagBits getAspect(VkFormat format) {
     if (format == VK_FORMAT_D24_UNORM_S8_UINT || format ==
VK_FORMAT_D16_UNORM ||
         format == VK_FORMAT_D16_UNORM_S8_UINT || format ==
VK_FORMAT_D32_SFLOAT ||
       format == VK_FORMAT_D32_SFLOAT_S8_UINT) {
      int ret;
      ret = VK_IMAGE_ASPECT_DEPTH_BIT;
      if (format >= VK_FORMAT_D16_UNORM_S8_UINT) {
        ret |= VK_IMAGE_ASPECT_STENCIL_BIT;
      }
      return static_cast<VkImageAspectFlagBits>(ret);
    } else {
      return VK_IMAGE_ASPECT_COLOR_BIT;
    }
  }

  void VulkanImageInterface::createDefaultImageView() {
                VkImageViewCreateInfo createInfo =
initializers::imageViewCreateInfo();
    createInfo.image = image();
    createInfo.format = imageInfo_.format;
    switch (imageInfo_.imageType) {
    case VK_IMAGE_TYPE_1D:
```

```
        createInfo.viewType = imageInfo_.arrayLayers > 1
                                  ? VK_IMAGE_VIEW_TYPE_1D_ARRAY
                                  : VK_IMAGE_VIEW_TYPE_1D;
        break;
      case VK_IMAGE_TYPE_2D:
        createInfo.viewType = imageInfo_.arrayLayers > 1
                                  ? VK_IMAGE_VIEW_TYPE_2D_ARRAY
                                  : VK_IMAGE_VIEW_TYPE_2D;
        break;
      case VK_IMAGE_TYPE_3D:
        createInfo.viewType = VK_IMAGE_VIEW_TYPE_3D;
        break;
      }

            createInfo.components    =    {VK_COMPONENT_SWIZZLE_R,
VK_COMPONENT_SWIZZLE_G,
                                           VK_COMPONENT_SWIZZLE_B,
VK_COMPONENT_SWIZZLE_A};
      createInfo.subresourceRange = {
          (VkImageAspectFlags)getAspect(createInfo.format), 0, 1,
0, 1};

        VK_CHECK_RESULT(vkCreateImageView(memoryManager->device(),
&createInfo,
                                                              nullptr,
&defaultImageView_))
    }

    VulkanImageInterface::~VulkanImageInterface() {
                      vkDestroyImageView(memoryManager->device(),
defaultImageView_, nullptr);
    }

    VkFormat convertInterfaceImageFormat(Image::Format format) {
      switch (format) {
      case Image::Format::RGBA8:
        return VK_FORMAT_R8G8B8A8_UNORM;
      case Image::Format::R11G11B10:
        return VK_FORMAT_B10G11R11_UFLOAT_PACK32;
      case Image::Format::D24S8:
        return VK_FORMAT_D24_UNORM_S8_UINT;
      default:
        return VK_FORMAT_UNDEFINED;
      }
    }
    VkImageType convertInterfaceImageType(Image::Type type) {
      switch (type) {
      case Image::Type::TEXTURE_1D:
        return VK_IMAGE_TYPE_1D;
      case Image::Type::TEXTURE_2D:
        return VK_IMAGE_TYPE_2D;
      case Image::Type::TEXTURE_3D:
```

```
        return VK_IMAGE_TYPE_3D;
      default:
        return VK_IMAGE_TYPE_MAX_ENUM;
      }
    }
    VkImageCreateInfo compileGeneralVulkanImageCI(ImageDesc desc)
{
      VkImageCreateInfo ret = initializers::imageCreateInfo();

      ret.usage = convertInterfaceImageUsageFlags(desc.usage);
      ret.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
      ret.format = convertInterfaceImageFormat(desc.format);

            if    (desc.type    !=    Image::Type::TEXTURE_3D    &&
desc.extents.depth != 1)
        throw std::runtime_error(
              "[VULKAN][ERROR] if Image::Type != TEXTURE_3D depth
must be equal 1");
            if    (desc.type    ==    Image::Type::TEXTURE_1D    &&
desc.extents.height != 1)
        throw std::runtime_error(
             "[VULKAN][ERROR] if Image::Type == TEXTURE_1D height
must be equal 1");

         ret.extent  =  {desc.extents.width,  desc.extents.height,
desc.extents.depth};
      ret.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
      ret.arrayLayers = desc.extents.layers;
      ret.samples = VK_SAMPLE_COUNT_1_BIT;
      ret.imageType = convertInterfaceImageType(desc.type);
        ret.tiling  =  desc.memType  ==  MemoryType::GPU_PRIVATE  ?
VK_IMAGE_TILING_OPTIMAL
                                                                  :
VK_IMAGE_TILING_LINEAR;
      ret.flags = 0;
      ret.mipLevels = 1;
      return ret;
    }

    VulkanGeneralImage::VulkanGeneralImage(VulkanMemoryManager
const *alloc,
                                           ImageDesc desc)
        : VulkanImage(alloc, desc.memType,
                      static_cast<VkBufferUsageFlagBits>(
                          convertInterfaceImageUsageFlags(desc.us
age)),
                      compileGeneralVulkanImageCI(desc)) {}

    VulkanStagingBuffer::VulkanStagingBuffer(const
VulkanMemoryManager *alloc,
                                             size_t size)
        : VulkanBuffer(alloc, MemoryType::HOST_COHERENT,
```

```
                        VK_BUFFER_USAGE_TRANSFER_SRC_BIT, size) {}
} // namespace APITest

APITest::VulkanBuffer::VulkanBuffer(VulkanMemoryManager const
*alloc,
                                    APITest::MemoryType type,
                                        VkBufferUsageFlagBits
usage, size_t size)
    : Allocatable(alloc), usage_(usage) {

  if (size == 0)
      throw std::runtime_error("[VULKAN][ERROR] trying to
create buffer with "
                               "size == 0 which is
prohibited.");

  size_ = size;
   // If allocating in private memory, pushing data will
involve explicit data
  // transfer operation
  if (type == MemoryType::GPU_PRIVATE)
    usage_ |= VK_BUFFER_USAGE_TRANSFER_DST_BIT;

                    VkBufferCreateInfo bufferCI =
initializers::bufferCreateInfo(usage_, size);
  bufferCI.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

                  VmaAllocationCreateInfo allocInfo =
chooseMemoryOptions(type);

      vmaCreateBuffer(memoryManager->allocator(), &bufferCI,
&allocInfo, &buffer_,
                      &allocation, &allocationInfo);
}

void APITest::VulkanBuffer::push(const void *data, size_t
size, size_t offset) {

  if (size + offset > allocationInfo.size)
    throw std::runtime_error(
        "[VULKAN][ERROR] pushing to out of bounds memory. " +
         std::to_string(size + offset - (allocationInfo.size))
+
        " bytes off bounds");

  const VkPhysicalDeviceMemoryProperties *pMemProps;
      vmaGetMemoryProperties(memoryManager->allocator(),
&pMemProps);
                    auto bits = pMemProps-
>memoryTypes[allocationInfo.memoryType].propertyFlags;

  if (bits & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ||
```

```
          bits & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT) {
        if (allocationInfo.pMappedData) {
            memcpy((char *)allocationInfo.pMappedData + offset,
data, size);
        } else {
          void *mapped;
            vmaMapMemory(memoryManager->allocator(), allocation,
&mapped);

          memcpy((char *)mapped + offset, data, size);

          vmaUnmapMemory(memoryManager->allocator(), allocation);
        }
      } else {
        VkBufferCreateInfo bufferCI =
            initializers::bufferCreateInfo(VK_BUFFER_USAGE_TRANSF
ER_SRC_BIT, size);
        bufferCI.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

        VmaAllocationCreateInfo allocInfo = {};
        allocInfo.usage = VMA_MEMORY_USAGE_CPU_ONLY;
                                  allocInfo.preferredFlags       =
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT |
                                    VK_MEMORY_PROPERTY_HOST_CACHED
_BIT;
                                  allocInfo.requiredFlags        =
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT;
        allocInfo.flags = VMA_ALLOCATION_CREATE_MAPPED_BIT;
        VmaAllocationInfo stagingBufferInfo;
        VmaAllocation stagingAllocation;
        VkBuffer stagingBuffer;

          vmaCreateBuffer(memoryManager->allocator(), &bufferCI,
&allocInfo,
                              &stagingBuffer, &stagingAllocation,
&stagingBufferInfo);

        auto mapped = stagingBufferInfo.pMappedData;

        memcpy((char *)mapped, data, size);

        VkBufferCopy copyRegion;
        copyRegion.size = size;
        copyRegion.dstOffset = offset;
        copyRegion.srcOffset = 0;
        memoryManager->copy(stagingBuffer, buffer_, copyRegion);

                       vmaDestroyBuffer(memoryManager->allocator(),
stagingBuffer,
                          stagingAllocation);
      }
    }
```

```
    APITest::VulkanMemoryManager::VulkanMemoryManager(
        APITest::VulkanRenderImpl *parent)
        : parent_(parent) {
      VmaAllocatorCreateInfo allocatorInfo = {};
      allocatorInfo.vulkanApiVersion = VK_API_VERSION_1_0;
       allocatorInfo.physicalDevice = parent_->getVulkanDevice()-
>getPhysical();
      allocatorInfo.device = parent_->getVulkanDevice()->get();
      allocatorInfo.instance = parent_->getInstance();

                VK_CHECK_RESULT(vmaCreateAllocator(&allocatorInfo,
&allocator_))
    }

    void     APITest::VulkanMemoryManager::copy(VkBuffer     src,
VkBuffer dst,
                                                VkBufferCopy region)
const {
      parent_->getQueueManager()->copyBuffer(src, dst, region);
    }

    APITest::VulkanMemoryManager::~VulkanMemoryManager() {
      vmaDestroyAllocator(allocator_);
    }

    VkDevice APITest::VulkanMemoryManager::device() const {
      return parent_->getVulkanDevice()->get();
    }

    void APITest::VulkanMemoryManager::transitLayout(
          APITest::VulkanImage *image, VkImageLayout newLayout)
const {
      VkCommandBuffer transitCmd;
                                      parent_->getQueueManager()-
>createPrimaryCommandBuffers(&transitCmd, 1, true);

      VkImageMemoryBarrier barrier{};
      barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
      barrier.oldLayout = image->currentLayout();
      barrier.newLayout = newLayout;

      barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
      barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

      barrier.image = image->image();
         barrier.subresourceRange.aspectMask  =  getAspect(image-
>format());
      barrier.subresourceRange.baseMipLevel = 0;
      barrier.subresourceRange.levelCount = 1;
      barrier.subresourceRange.baseArrayLayer = 0;
      barrier.subresourceRange.layerCount = 1;
```

```
        VkPipelineStageFlags sourceStage;
        VkPipelineStageFlags destinationStage;

        auto oldLayout = image->currentLayout();

        if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED &&
            newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
          barrier.srcAccessMask = 0;
          barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

          sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
          destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
                    }       else       if       (oldLayout       ==
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL &&
                                                    newLayout    ==
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
          barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
          barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

          sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
          destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
        } else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED &&
                                                    newLayout    ==
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
          barrier.srcAccessMask = 0;
          barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

          sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
          destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

        } else {
          assert(0 && "This image transition layout isn't supported
yet");
        }

        vkCmdPipelineBarrier(transitCmd, sourceStage /* TODO */,
                                  destinationStage /* TODO */, 0, 0,
nullptr, 0, nullptr,
                             1, &barrier);

        parent_->getQueueManager()->flush(
                            transitCmd,   parent_->getQueueManager()-
>getTransferQueue(), true);
      }

      void  APITest::VulkanMemoryManager::copy(APITest::VulkanBuffer
*buffer,
                                                 APITest::VulkanImage
*image) const {
        VkCommandBuffer copyCmd;
```

```
                                        parent_->getQueueManager()-
>createPrimaryCommandBuffers(&copyCmd, 1, true);

       VkBufferImageCopy region{};
       region.bufferOffset = 0;
       region.bufferRowLength = 0;
       region.bufferImageHeight = 0;

          region.imageSubresource.aspectMask   =   getAspect(image-
>format());
       region.imageSubresource.mipLevel = 0;
       region.imageSubresource.baseArrayLayer = 0;
       region.imageSubresource.layerCount = 1;

       region.imageOffset = {0, 0, 0};
       region.imageExtent = image->extents();

         vkCmdCopyBufferToImage(copyCmd,  buffer->buffer(),  image-
>image(),
                                VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
, 1, &region);

       parent_->getQueueManager()->flush(
                              copyCmd,    parent_->getQueueManager()-
>getTransferQueue(), true);
     }

     APITest::VulkanImage::VulkanImage(const
APITest::VulkanMemoryManager *alloc,
                                       APITest::MemoryType type,
                                       VkBufferUsageFlagBits usage,
                                       VkImageCreateInfo CI)
         : Allocatable(alloc), usage_(usage), imageInfo_(CI),
           currentLayout_(CI.initialLayout) {

                   VmaAllocationCreateInfo       allocInfo       =
chooseMemoryOptions(type);

         VK_CHECK_RESULT(vmaCreateImage(memoryManager->allocator(),
&CI, &allocInfo,
                                              &image_, &allocation,
&allocationInfo))
     }

     APITest::VulkanImage::~VulkanImage() {
             vmaDestroyImage(memoryManager->allocator(),    image_,
allocation);
     }
```

А.5 Лістинг програмної реалізації файлу VulkanMemoryManager.cpp

```
#include "VulkanPipeline.h"
#include "VulkanDevice.h"
#include "VulkanRenderImpl.h"
#include "vulkan_impl/VulkanDescriptorManager.h"
#include "vulkan_impl/VulkanRenderPass.h"
#include "vulkan_impl/VulkanShaderManager.h"
#include "vulkan_impl/util/VulkanInitializers.h"
#include "vulkan_impl/util/macro.h"
#include <array>
#include <glm/glm.hpp>

APITest::VulkanPipelineManager::VulkanPipelineManager(
    APITest::VulkanRenderImpl *parent)
    : parent_(parent) {
  VkPipelineCacheCreateInfo pipelineCacheCreateInfo = {};
                        pipelineCacheCreateInfo.sType           =
VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO;
                      VK_CHECK_RESULT(vkCreatePipelineCache(parent-
>getVulkanDevice()->get(),
                                                          &pipelineCacheCreateI
nfo, nullptr,
                                                         &pipelineCache));
}

APITest::VulkanPipelineManager::~VulkanPipelineManager() {
  auto device = parent_->getVulkanDevice()->get();

  vkDestroyPipelineCache(device, pipelineCache, nullptr);
}

static                                          std::pair<uint32_t,
std::vector<VkVertexInputAttributeDescription>>
vertexStateCI(APITest::GraphicsPipelineLayout const &desc) {
  uint32_t perVertexSize = 0;

                    std::vector<VkVertexInputAttributeDescription>
attributesDesc;

  auto &format = desc.vertexLayout.perVertexAttribute;

  for (auto attr : format) {
    switch (attr) {
    case APITest::VertexLayout::R32SF:
       attributesDesc.push_back(VkVertexInputAttributeDescript
ion(
           {static_cast<uint32_t>(attributesDesc.size()), 0,
            VK_FORMAT_R32_SFLOAT, perVertexSize}));
      perVertexSize += sizeof(float);
      break;
    case APITest::VertexLayout::RG32SF:
       attributesDesc.push_back(VkVertexInputAttributeDescript
ion(
```

```
                {static_cast<uint32_t>(attributesDesc.size()), 0,
                 VK_FORMAT_R32G32_SFLOAT, perVertexSize}));
            perVertexSize += sizeof(glm::vec2);
            break;
          case APITest::VertexLayout::RGB32SF:
             attributesDesc.push_back(VkVertexInputAttributeDescript
ion(
                {static_cast<uint32_t>(attributesDesc.size()), 0,
                 VK_FORMAT_R32G32B32_SFLOAT, perVertexSize}));
            perVertexSize += sizeof(glm::vec3);
            break;
          case APITest::VertexLayout::RGBA32SF:
             attributesDesc.push_back(VkVertexInputAttributeDescript
ion(
                {static_cast<uint32_t>(attributesDesc.size()), 0,
                 VK_FORMAT_R32G32B32A32_SFLOAT, perVertexSize}));
            perVertexSize += sizeof(glm::vec4);
            break;
          case APITest::VertexLayout::RGBA8UNORM:
             attributesDesc.push_back(VkVertexInputAttributeDescript
ion(
                {static_cast<uint32_t>(attributesDesc.size()), 0,
                 VK_FORMAT_R8G8B8A8_UNORM, perVertexSize}));
            perVertexSize += 4;
            break;
          case APITest::VertexLayout::MAT4F:
            for (int i = 0; i < 4; ++i) {
               attributesDesc.push_back(VkVertexInputAttributeDescri
ption(
                  {static_cast<uint32_t>(attributesDesc.size()), 0,
                   VK_FORMAT_R32G32B32A32_SFLOAT, perVertexSize}));
              perVertexSize += sizeof(glm::vec4);
            }
            break;
          }
        }

        return {perVertexSize, attributesDesc};
      }

      APITest::GraphicsPipelineRef
APITest::VulkanPipelineManager::get(
          APITest::GraphicsPipelineLayout const &desc) {
        return std::unique_ptr<GraphicsPipeline>(
            new VulkanGraphicsPipeline(this, desc));
      }

      APITest::VulkanGraphicsPipeline::VulkanGraphicsPipeline(
          APITest::VulkanPipelineManager *parent,
          APITest::GraphicsPipelineLayout layout)
          : VulkanPipeline(parent), layout_(std::move(layout)) {
```

```
      VkDescriptorSetLayout setLayout = VK_NULL_HANDLE;
                    if        (auto        *vkSetLayout        =
dynamic_cast<VulkanDescriptorSetLayout *>(
              layout_.descriptorsLayout.get())) {
        setLayout = vkSetLayout->layout();
      }
      VkPipelineLayoutCreateInfo pPipelineLayoutCreateInfo =
          initializers::pipelineLayoutCreateInfo(
              &setLayout, setLayout != VK_NULL_HANDLE ? 1 : 0);

      VK_CHECK_RESULT(vkCreatePipelineLayout(
                      parent_->parent_->getVulkanDevice()->get(),
&pPipelineLayoutCreateInfo,
          nullptr, &layoutHandle));
    }

    static VkCompareOp
    convertToVkCompareOp(APITest::RasterizerLayout::DepthTest::Co
mpOp compOp) {
      switch (compOp) {
                                                              case
APITest::RasterizerLayout::DepthTest::CompOp::LESS_OR_EQUAL:
        return VK_COMPARE_OP_LESS_OR_EQUAL;
      case APITest::RasterizerLayout::DepthTest::CompOp::LESS:
        return VK_COMPARE_OP_LESS;
      case APITest::RasterizerLayout::DepthTest::CompOp::GREATER:
        return VK_COMPARE_OP_GREATER;
                                                              case
APITest::RasterizerLayout::DepthTest::CompOp::GREATER_OR_EQUAL:
        return VK_COMPARE_OP_GREATER_OR_EQUAL;
      default:
        return VK_COMPARE_OP_MAX_ENUM;
      }
    }

    static VkCullModeFlags
    convertCullMode(APITest::RasterizerLayout::CullingState::Cull
Mode mode) {
      switch (mode) {
      case APITest::RasterizerLayout::CullingState::CULL_NONE:
        return VK_CULL_MODE_NONE;
                                                              case
APITest::RasterizerLayout::CullingState::CULL_FRONT_FACE:
        return VK_CULL_MODE_FRONT_BIT;
                                                              case
APITest::RasterizerLayout::CullingState::CULL_BACK_FACE:
        return VK_CULL_MODE_BACK_BIT;
      }
      return VK_CULL_MODE_FRONT_AND_BACK;
    }

    static VkFrontFace
```

```
    convertFrontFace(APITest::RasterizerLayout::CullingState::Fro
ntFace face) {
                                if              (face              ==
APITest::RasterizerLayout::CullingState::FRONT_FACE_CCW)
        return VK_FRONT_FACE_COUNTER_CLOCKWISE;
      else
        return VK_FRONT_FACE_CLOCKWISE;
    }
    VkPipeline
    APITest::VulkanGraphicsPipeline::get(APITest::VulkanRenderPas
s *pass) {

      auto *colorPass = dynamic_cast<VulkanColorPass *>(pass);

      if (!colorPass)
        throw std::runtime_error("[VULKAN][ERROR]: trying to bind
graphics "
                                 "pipeline to non-color render
pass.");

      auto vkPass = colorPass->get();
      if (perPassMap_.count(vkPass))
        return perPassMap_.at(vkPass);
      auto &instance = *parent_->parent_;

      VkPipelineInputAssemblyStateCreateInfo inputAssemblyState =
          initializers::pipelineInputAssemblyStateCreateInfo(
              VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, 0, VK_FALSE);
      VkPipelineRasterizationStateCreateInfo rasterizationState =
          initializers::pipelineRasterizationStateCreateInfo(
              VK_POLYGON_MODE_FILL,
               convertCullMode(layout_.rasterizerLayout.cullingSta
te.cullMode),
               convertFrontFace(layout_.rasterizerLayout.cullingSt
ate.face), 0);
      VkPipelineColorBlendAttachmentState blendAttachmentState{};
      if (layout_.blendingState.enable) {
        // Enable blending
        blendAttachmentState;
        blendAttachmentState.blendEnable = VK_TRUE;
        blendAttachmentState.colorWriteMask =
            VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
            VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
                  blendAttachmentState.srcColorBlendFactor    =
VK_BLEND_FACTOR_SRC_ALPHA;
        blendAttachmentState.dstColorBlendFactor =
            VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
        blendAttachmentState.colorBlendOp = VK_BLEND_OP_ADD;
        blendAttachmentState.srcAlphaBlendFactor =
            VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
                  blendAttachmentState.dstAlphaBlendFactor    =
VK_BLEND_FACTOR_ZERO;
```

```
        blendAttachmentState.alphaBlendOp = VK_BLEND_OP_ADD;

      } else
        blendAttachmentState =
            initializers::pipelineColorBlendAttachmentState(0xf,
VK_FALSE);

      VkPipelineColorBlendStateCreateInfo colorBlendState =
          initializers::pipelineColorBlendStateCreateInfo(1,
&blendAttachmentState);

      if (layout().rasterizerLayout.depthTest.enable &&
          !colorPass->hasDepthBuffer())
        throw std::runtime_error(
            "[VULKAN][ERROR]: binding graphics pipeline with
depth test enabled to "
            "color pass without depth buffer allocated.");

      VkPipelineDepthStencilStateCreateInfo depthStencilState =
          initializers::pipelineDepthStencilStateCreateInfo(
              layout().rasterizerLayout.depthTest.enable ?
VK_TRUE : VK_FALSE,
              layout().rasterizerLayout.depthTest.write ? VK_TRUE
: VK_FALSE,
              convertToVkCompareOp(layout().rasterizerLayout.dept
hTest.compOp));
      VkPipelineViewportStateCreateInfo viewportState =
          initializers::pipelineViewportStateCreateInfo(1, 1, 0);
      VkPipelineMultisampleStateCreateInfo multisampleState =
          initializers::pipelineMultisampleStateCreateInfo(VK_SAM
PLE_COUNT_1_BIT,
                                                           0);
          std::vector<VkDynamicState>    dynamicStateEnables    =
{VK_DYNAMIC_STATE_VIEWPORT,
                                                         VK_DYNAM
IC_STATE_SCISSOR};
      VkPipelineDynamicStateCreateInfo dynamicState =
          initializers::pipelineDynamicStateCreateInfo(dynamicSta
teEnables, 0);
          std::array<VkPipelineShaderStageCreateInfo,     2>
shaderStages{};
      VkGraphicsPipelineCreateInfo pipelineCI =
          initializers::pipelineCreateInfo(layoutHandle, vkPass,
0);
      pipelineCI.pInputAssemblyState = &inputAssemblyState;
      pipelineCI.pRasterizationState = &rasterizationState;
      pipelineCI.pColorBlendState = &colorBlendState;
      pipelineCI.pMultisampleState = &multisampleState;
      pipelineCI.pViewportState = &viewportState;
      pipelineCI.pDepthStencilState = &depthStencilState;
      pipelineCI.pDynamicState = &dynamicState;
      pipelineCI.stageCount = shaderStages.size();
```

```
        pipelineCI.pStages = shaderStages.data();

        auto [stride, attrDesc] = vertexStateCI(layout_);

        VkPipelineVertexInputStateCreateInfo ret{};

        VkVertexInputBindingDescription vertexAttrBindingInfo =
                        VkVertexInputBindingDescription({0,   stride,
VK_VERTEX_INPUT_RATE_VERTEX});

                                              ret.sType                    =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

        ret.vertexAttributeDescriptionCount = attrDesc.size();

        if (attrDesc.empty()) {
          ret.pVertexAttributeDescriptions = nullptr;
        } else {
          ret.pVertexAttributeDescriptions = attrDesc.data();
        }

        if (stride == 0) {
          ret.pVertexBindingDescriptions = nullptr;
          ret.vertexBindingDescriptionCount = 0;

        } else {
          ret.pVertexBindingDescriptions = &vertexAttrBindingInfo;
          ret.vertexBindingDescriptionCount = 1;
        }

        pipelineCI.pVertexInputState = &ret;
        pipelineCI.subpass = 0;

        // binding vertex shader

        shaderStages[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
        shaderStages[0].module =
                                            instance.getShaderManager()-
>getModule(layout_.vertexLayout.vertexShader);
                                  shaderStages[0].sType              =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        shaderStages[0].pName = "main";

        // binding fragment shader

        shaderStages[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
           shaderStages[1].module   =   instance.getShaderManager()-
>getModule(
            layout_.fragmentLayout.fragmentShader);
                                  shaderStages[1].sType              =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        shaderStages[1].pName = "main";
```

```
    VkPipeline newPipeline;

    VK_CHECK_RESULT(vkCreateGraphicsPipelines(
                    instance.getVulkanDevice()->get(),   parent_-
>pipelineCache, 1, &pipelineCI,
        nullptr, &newPipeline));

    perPassMap_.insert_or_assign(vkPass, newPipeline);

    return newPipeline;
  }

  APITest::VulkanGraphicsPipeline::~VulkanGraphicsPipeline() {}

  APITest::VulkanPipeline::~VulkanPipeline() {
    auto device = parent_->parent_->getVulkanDevice()->get();

    for (auto &pipeline : perPassMap_)
      vkDestroyPipeline(device, pipeline.second, nullptr);

    vkDestroyPipelineLayout(device, layoutHandle, nullptr);
  }
```

ДОДАТОК Б

Презентація

Актуальність проблеми

- Швидко зростаючий сфера 3D-графіки
- Великий потенціал для оптимізація для спеціалізованих задач та під конкретні графічні процесори
- Спеціалізовані цільові платформи

Сучасні API для роботи із графічним процесором

- DirectX
- OpenGL
- Vulkan API


DirectX
DirectX® 12
Цільова платформа – Windows
Зручний інтерфейс для роботи з примітивами
Відносно просте налаштування базового оточення
Відсутність доступу до низькорівневого налаштування процесів взаємоємодії з GPU
Ворона І.О., ст. гр. СКСм-20-1, каф. АПОТ, ХНУРЕ, 2021
5

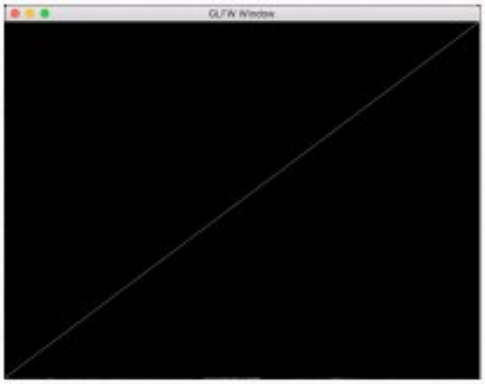

OpenGL
OpenGL
Цільові платформи: Linux, Windows
Перший відносно кросс платформений API для роботи з графікою
Великий набор бібліотек загального застосунку
Застарілий API відносно сучасних конкурентних API
Ворона І.О., ст. гр. СКСм-20-1, каф. АПОТ, ХНУРЕ, 2021
6

Vulkan API

- Цільові платформи: Linux, Windows
- Висока швидкість на усіх платформах
- Максимально низькорівневий інтерфейс для роботи з графічним процесором
- Потребує високий рівень знань користувача для використання

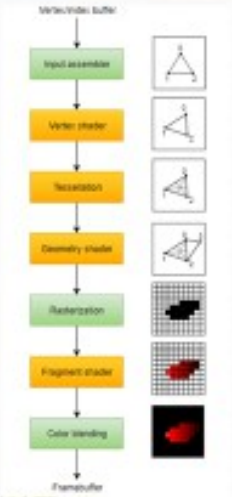
GLFW

GLFW (Graphics Library Framework) — модель для роботи з абстрактною віконною системою. GLFW надає можливість створювати кросс платформний користувацький віконний інтерфейс.



Графічний конвеєр

- Асемблер вхідних даних
- Шейдер вершин
- Теселяція
- Шейдер геометрії
- Растеризація
- Фрагментний шейдер



Реалізація моделі системи

```
APITest::VulkanGraphicsPipeline::VulkanGraphicsPipeline(
    APITest::VulkanPipelineManager *parent,
    APITest::GraphicsPipelineLayout layout)
    : VulkanPipeline(parent), layout_(std::move(layout)) {

  VkDescriptorSetLayout setLayout = VK_NULL_HANDLE;
  if (auto *vkSetLayout = dynamic_cast<VulkanDescriptorSetLayout *>(
          layout_.descriptorLayout.get())) {
    setLayout = vkSetLayout->layout();
  }
  VkPipelineLayoutCreateInfo pPipelineLayoutCreateInfo =
      initializers::pipelineLayoutCreateInfo(
          &setLayout, setLayout != VK_NULL_HANDLE ? 1 : 0);

  VK_CHECK_RESULT(vkCreatePipelineLayout(
      parent_->parent_->getVulkanDevice()->get(), &pPipelineLayoutCreateInfo,
      nullptr, &layoutHandle));
}
```

Створення графічного конвеєру:

- Опис GraphicsPipelineLayout
- Заповнення PipelineLayoutCreateInfo
- Перевірка конвеєру на драйвері GPU

Реалізація моделі системи

```
APITest::VulkanSwapChain::VulkanSwapChain(APITest::VulkanRendererImpl *parent,
                                          VkSurfaceKHR surface)
  : parent_(parent), surface_(surface) {

fpGetPhysicalDeviceSurfaceSupportKHR =
  reinterpret_cast<PFN_vkGetPhysicalDeviceSurfaceSupportKHR>(
    vkGetInstanceProcAddr(parent_->getInstance(),
                          "vkGetPhysicalDeviceSurfaceSupportKHR"));
fpGetPhysicalDeviceSurfaceCapabilitiesKHR =
  reinterpret_cast<PFN_vkGetPhysicalDeviceSurfaceCapabilitiesKHR>(
    vkGetInstanceProcAddr(parent_->getInstance(),
                          "vkGetPhysicalDeviceSurfaceCapabilitiesKHR"));
fpGetPhysicalDeviceSurfaceFormatsKHR =
  reinterpret_cast<PFN_vkGetPhysicalDeviceSurfaceFormatsKHR>(
    vkGetInstanceProcAddr(parent_->getInstance(),
                          "vkGetPhysicalDeviceSurfaceFormatsKHR"));
fpGetPhysicalDeviceSurfacePresentModesKHR =
  reinterpret_cast<PFN_vkGetPhysicalDeviceSurfacePresentModesKHR>(
    vkGetInstanceProcAddr(parent_->getInstance(), "vkGetPhysicalDeviceSurfacePresentModesKHR"));
```

Створення ланцюга кадрів:

- Опис робочої поверхні
- Аналіз доступних сімейств черг драйвера GPU
- Синхронізація елементів ланцюга за допомогою семафорів

Заключення

- Була розроблена модель обробки зображень на базі графічного API Vulkan;
- Реалізовано компоненти для роботи з рендерингом графічних примітивів та сцен з використанням OpenGL та Vulkan та допоміжних бібліотек glew та glm;
- Був проведен аналіз існуючих альтернативних рішень та API для роботи з графікою на різних операційних системах;
- Розроблена модель дає можливість вирішити проблему обробки та відображення 2D та 3D зображень, моделей і сцен.


Перспективи дослідження
Розширення користувацького інтерфейсу;
Подальший розвиток роботи із освітленням та текстуруванням;
Створення документації;
Оптимізація існуючої кодової бази для роботи на простіших GPU;
Розширення функціоналу при роботі із 3D моделями.
Ворона І.О., ст. гр. СКСм-20-1, каф. АПОТ, ХНУРЕ, 2021
13