Rendering Objects in Parallel Using Vulkan* APIs

EddieC (edwardcorreia78@gmail.com)
Praveen Kundurthy (praveen.k.kundurthy@intel.com)
Alexey Korenevsky (akorenevsky@ics.com)

If you're a game developer and not yet up to speed on Vulkan*, you should be. Vulkan APIs are one of the industry's hottest new technologies. They support multithreaded programming, simplify cross-platform development and have the backing of makers of major chips, GPUs, and devices. Vulkan APIs are positioned to become one of the next dominant graphics rendering platforms. Characteristics of the platform help apps gain longevity and run in more places. You might say that Vulkan lets apps live long and prosper—and this code sample will help get you started.

The APIs were introduced by the Khronos Group in 2015, and quickly gained the support of Intel and Google. Unity Technologies came on board in 2016, and Khronos confirmed plans to bestow Vulkan with support for multiple discrete GPUs automatically. By 2017, as the Vulkan APIs matured, an increasing number of game makers announced that they would begin adopting it. Vulkan became available for Apple's macOS* and iOS* platforms in 2018.

Vulkan carries a low overhead while also providing greater control over threading and memory management as well as improving direct access to the GPU over OpenGL* and other predecessor APIs. These features combine to give the developer versatility for targeting an array of platforms with essentially the same code base. With early backing from major industry players, the Vulkan platform has tremendous potential, and developers should be advised to get on board soon. Vulkan is built for now.

To help experienced pro and indie developers prepare for Vulkan, this article walks through the code of a sample app that renders multiple .fbx and .obj objects using Vulkan APIs. The app employs a non-touch graphical user interface (GUI) that reads and displays multiple object files in a common scene. Files are loaded and rendered using linear or parallel processing, selectable for the purpose of comparing performance. In addition, the app allows objects to be moved, rotated, and zoomed through a simple UI.

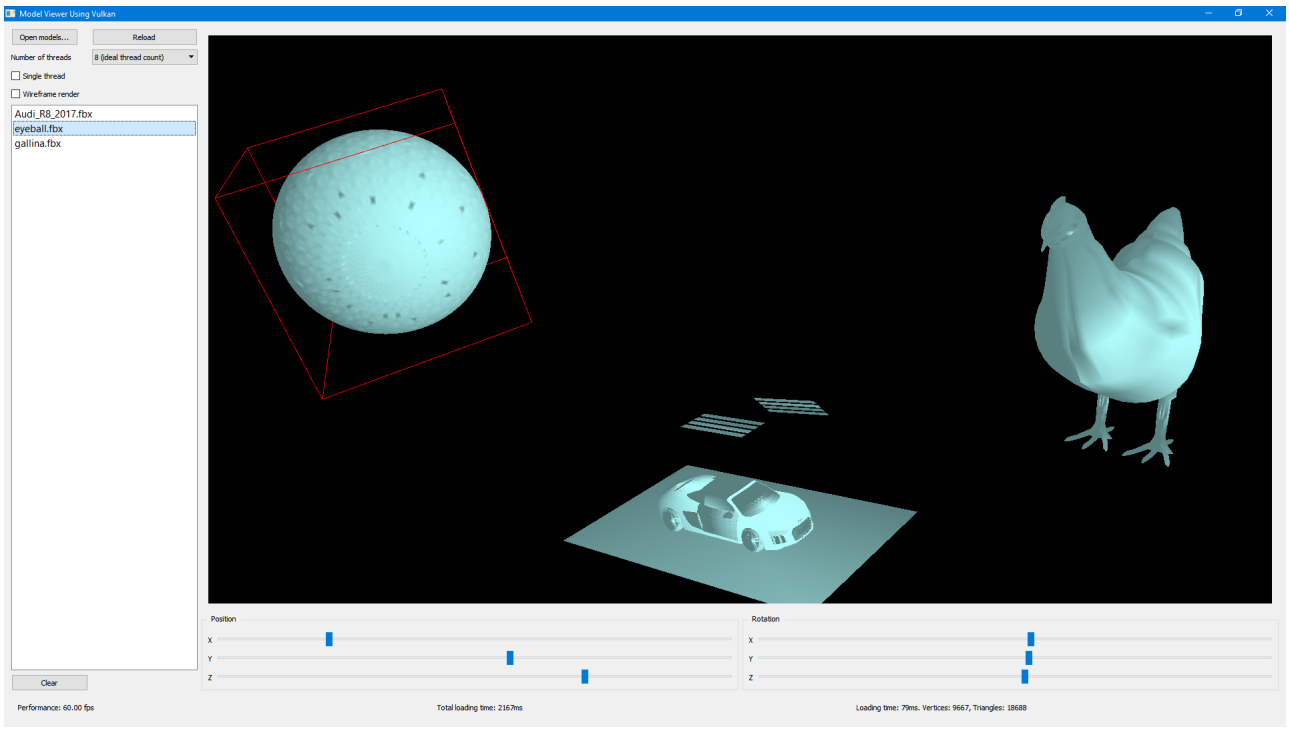

Figure 1: *Multiple rendered objects displayed simultaneously; the selected object is indicated with a bounding box.*

The app also features:

- Loaded models displayed in a list
- Selected objects identified on-screen with a bounding box
- An object info and stats display showing the number of vertices
- The ability to specify either delta or absolute coordinates and rotations
- Open object files in a file explorer window
- Option to view objects in wireframe mode
- Displays for stats for single- versus multithreading when reading and rendering

Keeping developers informed and educated on the latest technologies and development techniques is an important part of ensuring their success and prosperity. To that end, all source code and libraries from this project are available for download, so you can build and learn from the app on your own and adapt the functions for use in your own apps.

For people new to Vulkan, the learning curve could be steep. Because it gives developers rich features and a broad level of control, Vulkan contains far more structures and requires a greater number of initializations than OpenGL and other graphics libraries. For the sample app, the renderer alone (renderer.cpp) required more than 500 lines of code.

In an effort to minimize the amount of code required, this sample app focuses heavily on architecting a unified means of rendering different object types. Commonalities are identified in the initialization steps, which are separated from the general pipeline, and parts specific to a particular instance of 3D objects are loaded and rendered from a file. A boundary box is another type of object and requires its own shaders, settings, and pipeline. There is only one instance,

however. Minimizing coding differences between object types also helped to improve flexibility and simplify the code.

One of the most significant challenges of developing this sample involved multithreaded rendering. Though Vulkan APIs are considered "thread-safe," some objects required explicit synchronization on the host side and at the point of implementation if applied to command pool and command buffer. When an object requests the command buffer, the buffer is allocated from the command pool. If the command pool is accessed in parallel from several threads at once, the app would crash or report a warning in the Vulkan console. One answer would be to use mutual exclusions, or mutexes, to serialize an access to the shared command pool. But this would eliminate the advantage of parallel processing, because threads would compete and block each other. Instead, the sample app implements separate command buffers and command pools for each 3D object instance, which then requires extra code for the release of resources.

What You'll Need

The minimum requirement for developing with Vulkan APIs on Intel® Graphics Processing Units (GPUs) is a processor from the 6th Generation Intel® processor family (introduced in August 2015), running 64-bit Windows® 7, 8.1, or 10. Intel also offers a 64-bit Windows® 10-only driver for 6th, 7th, or 8th Generation processors. Vulkan drivers are now included with Intel® HD Graphics drivers, which helps simplify the setup process. Instructions are available for installing Vulkan drivers on Intel®-based systems running Unity* or Unreal* Engine 4.

Code Walk-Through

This app was built as an aid to developers learning to use Vulkan. This walk-through explains the techniques used to make the sample app, simplifying the work of getting started on your own. To reduce time spent on planning the architecture, the app was developed using an incremental, iterative process, which helps minimize changes during the coding phase. The project was divided into three parts: UI (MainWindow and VulkanWindow), model loader (Model.cpp/h), and rendering (Renderer.cpp/h). The feature list was prioritized and sorted by difficulty of implementation. Coding then started with the easiest features—refactoring and changing design only when needed.

MainWindow.cpp

In the sample app's main window, object files are loaded using either a single process or in parallel. Either way, a timer counts the total loading time to allow for comparison. When files are processed in parallel, the QtConcurrent component is used to process worker threads.

The "loadModels()" function starts the parallel or linear processing of files. In the first few lines, a counter is started. Then the loading times for file(s) are counted and an aiScene is created using the Asimp external library. Next, the aiScene is converted to a class model created for this app that's more convenient to Vulkan. A progress dialog is created and presented while parallel file processing takes place.

```
void MainWindow::loadModels()
```

```
{
   clearModels();
   m_elapsedTimer.start(); // counts total loading time

   std::function<QSharedPointer<Model>(const QString &)> load = [](const QString &path) {
      QSharedPointer<Model> model;
      QFileInfo info(path);
      if (!info.exists())
         return model;
      QElapsedTimer timer;
      timer.start(); // loading time for this file
      Assimp::Importer importer;
// read file from disk and create aiScene (external library Asimp) instance
      const aiScene* scene = importer.ReadFile(path.toStdString(),
                                    aiProcess_Triangulate |
                                    aiProcess_RemoveComponent |
                                    aiProcess_GenNormals |
                                    aiProcess_JoinIdenticalVertices);

      qDebug() << path << (scene ? "OK" : importer.GetErrorString());
      if (scene) {
// aiScene format is not very convenient for renderer so we designed class Model to keep data ready for Vulkan renderer.
         model = QSharedPointer<Model>::create(info.fileName(), scene);  //convert aiScene to class Model (Model.cpp) that's convenient for Vulkan renderer

         if (model->isValid()) {
            model->loadingTime = timer.elapsed();
         } else {
            model.clear();
         }
      }
      return model;
   };
// create a progress dialog for app user
   if (m_progressDialog == nullptr) {
      m_progressDialog = new QProgressDialog(this);
      QObject::connect(m_progressDialog, &QProgressDialog::canceled, &m_loadWatcher, &QFutureWatcher<void>::cancel);
      QObject::connect(&m_loadWatcher,  &QFutureWatcher<void>::progressRangeChanged, m_progressDialog, &QProgressDialog::setRange);
      QObject::connect(&m_loadWatcher, &QFutureWatcher<void>::progressValueChanged,  m_progressDialog, &QProgressDialog::setValue);
   }
```

```
   // using QtConcurrent for parallel file processing in worker threads
   QFuture<QSharedPointer<Model>> future = QtConcurrent::mapped(m_files, load);
   m_loadWatcher.setFuture(future);
//present the progress dialog to app user
   m_progressDialog->exec();
}
```

The “loadFinished()” function processes results of the parallel or linear processing, adds object file names to “listView,” and passes models to the renderer.

```
void MainWindow::loadFinished() {
   qDebug("loadFinished");
   Q_ASSERT(m_vulkanWindow->renderer());
   m_progressDialog->close(); // close the progress dialog
// iterate around result of file load
   const auto & end = m_loadWatcher.future().constEnd();

// loop for populating list of file names
   for (auto it = m_loadWatcher.future().constBegin() ; it != end; ++it) {
      QSharedPointer<Model> model = *it;
      if (model) {
         ui->modelsList->addItem(model->fileName); // populates list view
// pass object to renderer (created in vulkanWindow, which is part of the mainWindow)
         m_vulkanWindow->renderer()->addObject(model);
      }
   }
```

Identify the selected object on the screen by surrounding it with a bounding box.

```
mainwindow.cpp: MainWindow::currentRowChanged(int row)
{
...
  if (m_vulkanWindow->renderer())
        m_vulkanWindow->renderer()->selectObject(row);

renderer.cpp: Renderer::selectObject(int index) - inflates BoundaryBox object’s  model
...
```

Display object info and statistics (i.e., number of vertices) of the selected object on the screen. Here, object-specific statistics are created and loading time for the scene is displayed.

```
MainWindow::currentRowChanged(int row) - shows statistic for selected object:
{
…
// prepare object-specific statistics (verticies, etc)
QString stat = tr("Loading time: %1ms. Vertices: %2, Triangles: %3")
```

```
            .arg(item->model->loadingTime)
            .arg(item->model->totalVerticesCount())
            .arg(item->model->totalTrianglesCount());
ui->objectStatLabel->setText(stat);

// display total scene loading time
void MainWindow::loadFinished()
            ui->totalStatLabel->setText(tr("Total loading time: %1ms").arg(m_elapsedTimer.elapsed()));

// show rendering performance in frames per second
void MainWindow::timerEvent(QTimerEvent *)
            ui->fpsLabel->setText(tr("Performance: %1 fps").arg(renderer->fps(), 0, 'f', 2, '0'));
...
```

Enable users of the app to specify absolute coordinates and rotations.

```
void MainWindow::positionSliderChanged(int)
{
   const int row = ui->modelsList->currentRow();
   if (row == -1 || m_ignoreSlidersSignal || !m_vulkanWindow->renderer())
      return;
   m_vulkanWindow->renderer()->setPosition(row, ui->posXSlider->value() / 100.0f, ui->posYSlider->value() / 100.0f,
                        ui->posZSlider->value() / 100.0f );
}

void MainWindow::rotationSliderChanged(int)
{
   const int row = ui->modelsList->currentRow();
   if (row == -1 || m_ignoreSlidersSignal || !m_vulkanWindow->renderer())
      return;
    m_vulkanWindow->renderer()->setRotation(row, ui->rotationXSlider->value(), ui->rotationYSlider->value(),
                        ui->rotationZSlider->value());
}
```

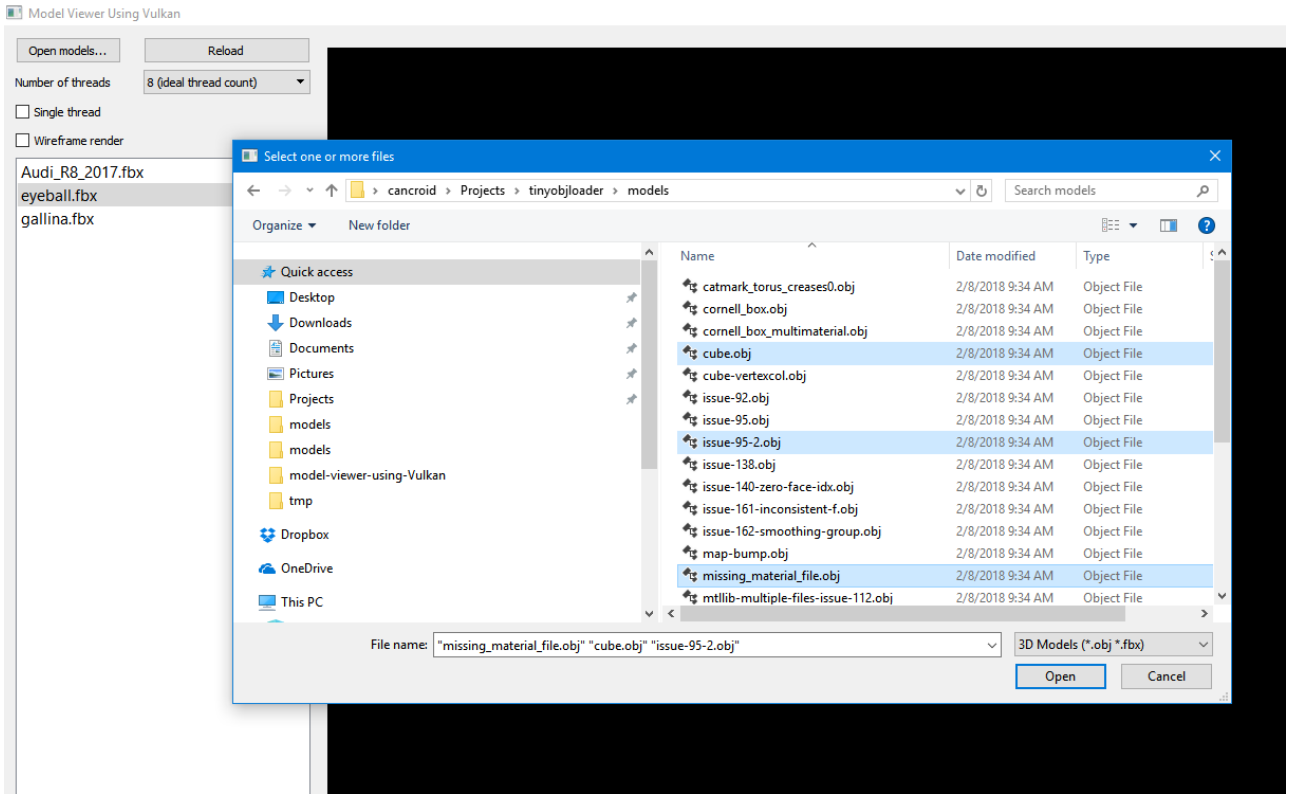

Figure 2: *The sample app implements a file explorer window for finding and opening objects to render.*

Allow the app to open object files using a file explorer window.

```
MainWindow::MainWindow(QWidget *parent)
   : QWidget(parent),
     ui(new Ui::MainWindow)
{
...

connect(ui->loadButton, &QPushButton::clicked, this, [this] {
      const QStringList & files = QFileDialog::getOpenFileNames(this, tr("Select one or more files"), QString::null, "3D
Models (*.obj *.fbx)");
      if (!files.isEmpty()) {
         m_files = files;
         loadModels();
         ui->reloadButton->setEnabled(true);
      }
  });

...
```

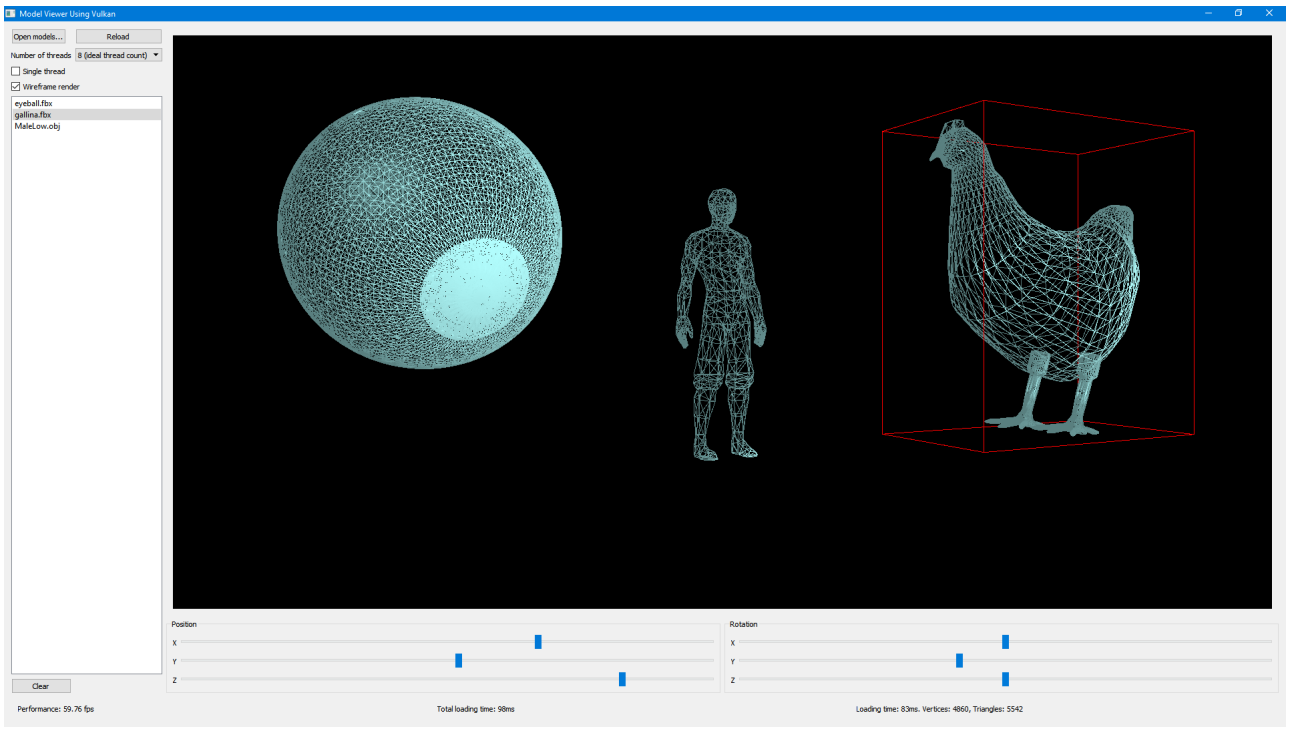

Figure 3: *Objects rendered in wireframe mode; the selected object is indicated by a bounding box.*

Allow the user to display objects in wireframe mode.

```
MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent),
      ui(new Ui::MainWindow)
{
...
 connect(ui->wireframeSwitch, &QCheckBox::stateChanged, this, [this]{
      if (m_vulkanWindow->renderer()) {
         m_vulkanWindow->renderer()->setWirefameMode(ui->wireframeSwitch->checkState() == Qt::Checked);
      }
   });
```

Renderer.cpp (line 386-402):

```
void Renderer::setWirefameMode(bool enabled)

...
```

Renderer.cpp

Because of the complexities of the Vulkan APIs, the biggest challenge to this app's developer was building Renderer, which implements application-specific rendering logic for VulkanWindow.

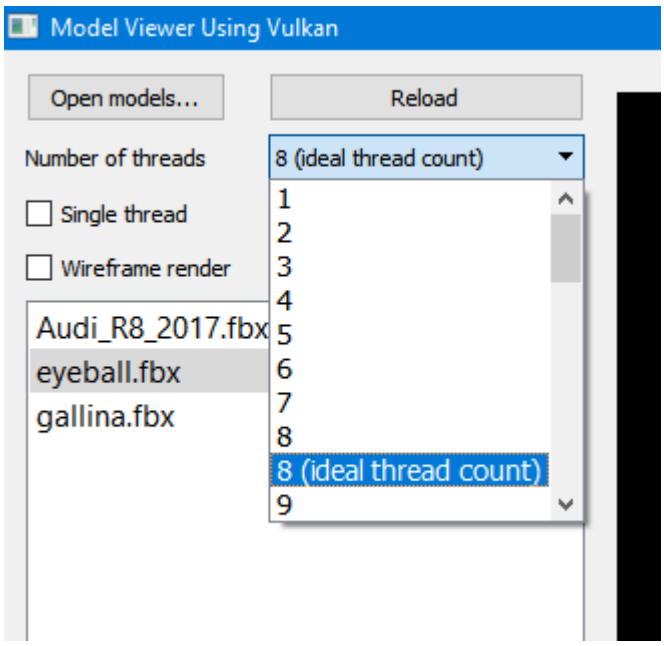

Figure 4: *Thread selection is simplified with a drop-down window; the ideal number is based on cores in the host system.*

Especially challenging was the synchronization of worker and UI threads without using mutual exclusive locks on rendering and resource releasing phases. On the rendering phase, this is achieved by separating command pools and secondary command buffers for each Object3D instance. In the resource releasing phase, it is necessary to make sure the host and GPU rendering phases are finished.

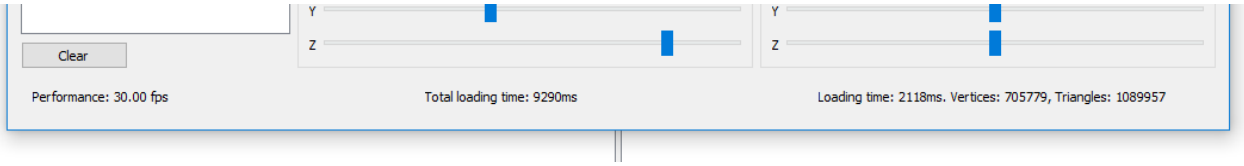

Figure 5: *Total loading time and vertices count of an object file allow comparison of single- and multithreaded loading times.*

Rendering Results May Vary

The system processor, GPU, and other factors of the host system as well as the size of the object file will determine single- and multithreaded object rendering times. Your results will vary. Normally, the host rendering phase is finished when “Renderer::&m_renderWatcher” emits a “finished” signal and “Renderer::endFrame()” is called. The resource-releasing phase might be initiated in cases such as:

1. The Vulkan window is resized or closed. “Renderer::releaseSwapChainResources” and “Renderer::releaseResources” will be called.
2. The wireframe mode changed—"Renderer::setWirefameMode”
3. Objects are deleted—“Renderer::deleteObjects”
4. Objects are added—“Renderer::addObject”

In those situations, the first things we need to do are:

1. Wait until all worker threads are finished.
2. Explicitly finish the rendering phase, calling “Renderer::endFrame()”, which also sets the flag “m_framePreparing = false” to ignore all results from worker threads that come asynchronously in the near future.

3. Wait until the GPU finishes all graphical queues using the “m_deviceFunctions->vkDeviceWaitIdle(m_window->device())” call.

This is implemented in “Renderer::rejectFrame”:

```
void Renderer::rejectFrame()
{
  m_renderWatcher.waitForFinished(); // all workers must be finished
  endFrame(); // flushes current frame
  m_deviceFunctions->vkDeviceWaitIdle(m_window->device()); // all graphics queues must be finished
}
```

Parallel preparation of command buffers to render 3D objects iseature is implemented in the following three functions; the code for each follows after:

1. Renderer::startNextFrame—This is called when the draw commands for the current frame need to be added.
2. Renderer::drawObject—This records commands to the secondary command buffer. This is running in worker thread. When it’s done, the buffer is reported to the UI thread to be recorded to the primary command buffer.
3. Renderer::endFrame—This finishes the render pass for current command buffer, reports to VulkanWindow that a frame is ready, and requests an immediate update to keep rendering.

Function 1: void Renderer::startNextFrame()

This section contains mainly Vulkan-specific code that is not likely to need modification. The snippet is intended to show how to load an object file using Vulkan. About a dozen lines in, the loaded file is sent to the renderer with support for a secondary command buffer to allow object-loading in parallel.

```
void Renderer::startNextFrame()
{
  m_framePreparing = true;

  const QSize imageSize = m_window->swapChainImageSize();

  VkClearColorValue clearColor = { 0, 0, 0, 1 };

  VkClearValue clearValues[3] = {};
  clearValues[0].color = clearValues[2].color = clearColor;
  clearValues[1].depthStencil = { 1, 0 };

  VkRenderPassBeginInfo rpBeginInfo = {};
  memset(&rpBeginInfo, 0, sizeof(rpBeginInfo));
  rpBeginInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
  rpBeginInfo.renderPass = m_window->defaultRenderPass();
  rpBeginInfo.framebuffer = m_window->currentFramebuffer();
  rpBeginInfo.renderArea.extent.width = imageSize.width();
  rpBeginInfo.renderArea.extent.height = imageSize.height();
```

```
    rpBeginInfo.clearValueCount = m_window->sampleCountFlagBits() > VK_SAMPLE_COUNT_1_BIT ? 3 : 2;
    rpBeginInfo.pClearValues = clearValues;

    // starting render pass with secondary command buffer support
    m_deviceFunctions->vkCmdBeginRenderPass(m_window->currentCommandBuffer(), &rpBeginInfo,
VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS);

    if (m_objects.size()) {
        // starting parallel command buffers generation in worker threads using QtConcurrent
        auto drawObjectFn = std::bind(&Renderer::drawObject, this, std::placeholders::_1);
        QFuture<VkCommandBuffer> future = QtConcurrent::mapped(m_objects, drawObjectFn);
        m_renderWatcher.setFuture(future);
} else {
// if no object exists, end immediately
        endFrame();
    }
}
```

Function 2: Renderer::endFrame()

This function instructs Vulkan that all command buffers are ready for rendering with the GPU.

```
void Renderer::endFrame()
{
    if (m_framePreparing) {
        m_framePreparing = false;
        m_deviceFunctions->vkCmdEndRenderPass(m_window->currentCommandBuffer());
        m_window->frameReady();
        m_window->requestUpdate();
        ++m_framesCount;
    }
}
```

Function 3: Renderer::drawObject()

The function prepares the command buffers to be sent to the GPU. As above, the Vulkan-specific code in this snippet also runs in a worker thread and is not likely to need modification for use in other apps.

```
// running in a worker thread
VkCommandBuffer Renderer::drawObject(Object3D * object)
{
    if (!object->model)
        return VK_NULL_HANDLE;
```

```
  const PipelineHandlers & pipelineHandlers = object->role == Object3D::Object ? m_objectPipeline :
m_boundaryBoxPipeline;
  VkDevice device = m_window->device();

  if (object->vertexBuffer == VK_NULL_HANDLE) {
    initObject(object);
  }

  VkCommandBuffer & cmdBuffer = object->cmdBuffer[m_window->currentFrame()];

  VkCommandBufferInheritanceInfo inherit_info = {};
  inherit_info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO;
  inherit_info.renderPass = m_window->defaultRenderPass();
  inherit_info.framebuffer = m_window->currentFramebuffer();

  VkCommandBufferBeginInfo cmdBufBeginInfo = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    nullptr,
    VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT,
    &inherit_info
  };
  VkResult res = m_deviceFunctions->vkBeginCommandBuffer(cmdBuffer, &cmdBufBeginInfo);
  if (res != VK_SUCCESS) {
    qWarning("Failed to begin frame command buffer: %d", res);
    return VK_NULL_HANDLE;
  }

  const QSize & imageSize = m_window->swapChainImageSize();

  VkViewport viewport;
  viewport.x = viewport.y = 0;
  viewport.width = imageSize.width();
  viewport.height = imageSize.height();
  viewport.minDepth = 0;
  viewport.maxDepth = 1;
  m_deviceFunctions->vkCmdSetViewport(cmdBuffer, 0, 1, &viewport);

  VkRect2D scissor;
  scissor.offset.x = scissor.offset.y = 0;
  scissor.extent.width = imageSize.width();
  scissor.extent.height = imageSize.height();
  m_deviceFunctions->vkCmdSetScissor(cmdBuffer, 0, 1, &scissor);
```

```
    QMatrix4x4 objectMatrix;
    objectMatrix.translate(object->translation.x(), object->translation.y(), object->translation.z());
    objectMatrix.rotate(object->rotation.x(), 1, 0, 0);
    objectMatrix.rotate(object->rotation.y(), 0, 1, 0);
    objectMatrix.rotate(object->rotation.z(), 0, 0, 1);
    objectMatrix *= object->model->transformation;



    m_deviceFunctions->vkCmdBindPipeline(cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
pipelineHandlers.pipeline);

    // pushing view-projection matrix to constants
    m_deviceFunctions->vkCmdPushConstants(cmdBuffer, pipelineHandlers.pipelineLayout,
VK_SHADER_STAGE_VERTEX_BIT, 0, 64, m_world.constData());

    const int nodesCount = object->model->nodes.size();
    for (int n = 0; n < nodesCount; ++n) {
        const Node &node = object->model->nodes.at(n);
        const uint32_t frameUniSize = nodesCount * object->uniformAllocSize;
        const uint32_t frameUniOffset = m_window->currentFrame() * frameUniSize + n * object->uniformAllocSize;
        m_deviceFunctions->vkCmdBindDescriptorSets(cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
pipelineHandlers.pipelineLayout, 0, 1,
                                   &object->descSet, 1, &frameUniOffset);

        // mapping uniform buffer to update matrix
        quint8 *p;
        res = m_deviceFunctions->vkMapMemory(device, object->bufferMemory, object->uniformBufferOffset +
frameUniOffset,
                                             MATRIX_4x4_SIZE, 0, reinterpret_cast<void **>(&p));
        if (res != VK_SUCCESS)
            qFatal("Failed to map memory: %d", res);

        QMatrix4x4 nodeMatrix = objectMatrix * node.transformation;
        memcpy(p, nodeMatrix.constData(), 16 * sizeof(float)); //updating matrix

        m_deviceFunctions->vkUnmapMemory(device, object->bufferMemory);

        // drawing meshes
        for (const int i: qAsConst(node.meshes)) {
            const Mesh &mesh = object->model->meshes.at(i);
            VkDeviceSize vbOffset = mesh.vertexOffsetBytes();
            m_deviceFunctions->vkCmdBindVertexBuffers(cmdBuffer, 0, 1, &object->vertexBuffer, &vbOffset);
```

```
        m_deviceFunctions->vkCmdBindIndexBuffer(cmdBuffer, object->vertexBuffer, object->indexBufferOffset +
mesh.indexOffsetBytes(), VK_INDEX_TYPE_UINT32);


        m_deviceFunctions->vkCmdDrawIndexed(cmdBuffer, mesh.indexCount, 1, 0, 0, 0);
      }
   }


   m_deviceFunctions->vkEndCommandBuffer(cmdBuffer);


   return cmdBuffer;
}
```

The complete secondary buffer is reported back to a GUI thread, and commands can be executed on the primary buffer (unless frame rendering is canceled):

Renderer.cpp (line 31-38):

```
QObject::connect(&m_renderWatcher, &QFutureWatcher<VkCommandBuffer>::resultReadyAt, [this](int index) {
     // secondary command buffer of some object is ready
     if (m_framePreparing) {
        const VkCommandBuffer & cmdBuf = m_renderWatcher.resultAt(index);
        if (cmdBuf)
           this->m_deviceFunctions->vkCmdExecuteCommands(this->m_window->currentCommandBuffer(), 1,
&cmdBuf);
     }
  });
...
```

Another major challenge to development of the renderer came in the handling of different types of graphical objects—those loaded from files and those dynamically generated in the form of boundary boxes that surround selected objects. This caused a problem because they use differing shaders, primitive topologies, and polygon modes. The goal was to unify code, as much as possible, for different objects to avoid replication of similar code. Both types of objects are expressed by single-class Object3D.

In the “Renderer::initPipelines()” function, differences were isolated as function parameters and called in this way:

```
initPipeline(m_objectPipeline,
             QStringLiteral(":/shaders/item.vert.spv"),
              QStringLiteral(":/shaders/item.frag.spv"),
             VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,
             m_wireframeMode ? VK_POLYGON_MODE_LINE : VK_POLYGON_MODE_FILL);

initPipeline(m_boundaryBoxPipeline,
             QStringLiteral(":/shaders/selection.vert.spv"),
             QStringLiteral(":/shaders/selection.frag.spv"),
             VK_PRIMITIVE_TOPOLOGY_LINE_LIST, VK_POLYGON_MODE_LINE);
```

It also proved helpful to unify initialization of particular objects according to their role. This is handled by the “Renderer::initObject()” function:

```
const PipelineHandlers & pipelineHandlers = object->role == Object3D::Object ? m_objectPipeline :
m_boundaryBoxPipeline;
```

"Function: Renderer::initPipeline()" shows the full function. Note that in addition to object files, the boundary box is another type of object and requires its own shaders, settings, and pipeline. Minimizing coding differences between object types also helped to improve flexibility and simplify the code.

```
void Renderer::initPipeline(PipelineHandlers & pipeline, const QString & vertShaderPath, const QString &
fragShaderPath,
                    VkPrimitiveTopology topology, VkPolygonMode polygonMode)
{
  VkDevice device = m_window->device();
  VkResult res;
  VkVertexInputBindingDescription vertexBindingDesc = {
    0, // binding
    6 * sizeof(float), //x,y,z,nx,ny,nz
    VK_VERTEX_INPUT_RATE_VERTEX
  };

  VkVertexInputAttributeDescription vertexAttrDesc[] = {
    { // vertex
      0,
      0,
      VK_FORMAT_R32G32B32_SFLOAT,
      0
    },
    { // normal
      1,
      0,
      VK_FORMAT_R32G32B32_SFLOAT,
      6 * sizeof(float)
    }
  };


  VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
  vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
  vertexInputInfo.vertexBindingDescriptionCount = 1;
  vertexInputInfo.pVertexBindingDescriptions = &vertexBindingDesc;
  vertexInputInfo.vertexAttributeDescriptionCount = 2;
  vertexInputInfo.pVertexAttributeDescriptions = vertexAttrDesc;
```

```
  VkDescriptorSetLayoutBinding layoutBinding = {};
  layoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC;
  layoutBinding.descriptorCount = 1;
  layoutBinding.stageFlags =  VK_SHADER_STAGE_VERTEX_BIT;

  VkDescriptorSetLayoutCreateInfo descLayoutInfo = {
     VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
     nullptr,
     0,
     1,
     &layoutBinding
  };

   //!  View-projection matrix going to be pushed to vertex shader constants.
  VkPushConstantRange push_constant = {
        VK_SHADER_STAGE_VERTEX_BIT,
        0,
        64
     };

  res = m_deviceFunctions->vkCreateDescriptorSetLayout(device, &descLayoutInfo, nullptr,
&pipeline.descSetLayout);
  if (res != VK_SUCCESS)
     qFatal("Failed to create descriptor set layout: %d", res);



  // Pipeline layout
  VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
  pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  pipelineLayoutInfo.setLayoutCount = 1;
  pipelineLayoutInfo.pSetLayouts = &pipeline.descSetLayout;
  pipelineLayoutInfo.pushConstantRangeCount = 1;
  pipelineLayoutInfo.pPushConstantRanges = &push_constant;

  res = m_deviceFunctions->vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipeline.pipelineLayout);
  if (res != VK_SUCCESS)
     qFatal("Failed to create pipeline layout: %d", res);

  // Shaders
  VkShaderModule vertShaderModule = loadShader(vertShaderPath);
```

```
VkShaderModule fragShaderModule = loadShader(fragShaderPath);

// Graphics pipeline
VkGraphicsPipelineCreateInfo pipelineInfo;
memset(&pipelineInfo, 0, sizeof(pipelineInfo));
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;

VkPipelineShaderStageCreateInfo shaderStageCreationInfo[2] = {
  {
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,
    nullptr,
    0,
    VK_SHADER_STAGE_VERTEX_BIT,
    vertShaderModule,
    "main",
    nullptr
  },
  {
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,
    nullptr,
    0,
    VK_SHADER_STAGE_FRAGMENT_BIT,
    fragShaderModule,
    "main",
    nullptr
  }
};
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStageCreationInfo;

pipelineInfo.pVertexInputState = &vertexInputInfo;

VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo = {};
inputAssemblyInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssemblyInfo.topology = topology;
pipelineInfo.pInputAssemblyState = &inputAssemblyInfo;

VkPipelineViewportStateCreateInfo viewportInfo = {};
viewportInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
viewportInfo.viewportCount = 1;
viewportInfo.scissorCount = 1;
pipelineInfo.pViewportState = &viewportInfo;
```

```
  VkPipelineRasterizationStateCreateInfo rasterizationInfo = {};
  rasterizationInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
  rasterizationInfo.polygonMode = polygonMode;
  rasterizationInfo.cullMode = VK_CULL_MODE_NONE;
  rasterizationInfo.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
  rasterizationInfo.lineWidth = 1.0f;
  pipelineInfo.pRasterizationState = &rasterizationInfo;

  VkPipelineMultisampleStateCreateInfo multisampleInfo = {};
  multisampleInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
  multisampleInfo.rasterizationSamples = m_window->sampleCountFlagBits();
  pipelineInfo.pMultisampleState = &multisampleInfo;

  VkPipelineDepthStencilStateCreateInfo depthStencilInfo = {};
  depthStencilInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
  depthStencilInfo.depthTestEnable = VK_TRUE;
  depthStencilInfo.depthWriteEnable = VK_TRUE;
  depthStencilInfo.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
  pipelineInfo.pDepthStencilState = &depthStencilInfo;

  VkPipelineColorBlendStateCreateInfo colorBlendInfo  = {};
  colorBlendInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
  VkPipelineColorBlendAttachmentState att = {};
  att.colorWriteMask = 0xF;
  colorBlendInfo.attachmentCount = 1;
  colorBlendInfo.pAttachments = &att;
  pipelineInfo.pColorBlendState = &colorBlendInfo;

  VkDynamicState dynamicEnable[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };
  VkPipelineDynamicStateCreateInfo dynamicInfo = {};
  dynamicInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
  dynamicInfo.dynamicStateCount = 2;
  dynamicInfo.pDynamicStates = dynamicEnable;
  pipelineInfo.pDynamicState = &dynamicInfo;

  pipelineInfo.layout = pipeline.pipelineLayout;
  pipelineInfo.renderPass = m_window->defaultRenderPass();

  res = m_deviceFunctions->vkCreateGraphicsPipelines(device, m_pipelineCache, 1, &pipelineInfo, nullptr,
&pipeline.pipeline);
  if (res != VK_SUCCESS)
```

```
    qFatal("Failed to create graphics pipeline: %d", res);

  if (vertShaderModule)
    m_deviceFunctions->vkDestroyShaderModule(device, vertShaderModule, nullptr);
  if (fragShaderModule)
    m_deviceFunctions->vkDestroyShaderModule(device, fragShaderModule, nullptr);
}
```

Conclusion

Coding flexibility is a hallmark of low-level Vulkan APIs, but it's critical to remain focused on what's going on in each Vulkan step. Lower-level programming also allows for precise fine-tuning of certain aspects of hardware access not available with OpenGL. If you take it slow and build your project in small, incremental steps, the payoffs will include far greater rendering performance, much lower runtime footprint, and greater portability to a multitude of devices and platforms.

Pros and indies alike should prepare for Vulkan. This article provided a walk-through of an app that shows how to use Vulkan APIs to render multiple .fbx and .obj objects, and read and display multiple object files in a common scene. You've also seen how to integrate a file explorer window to load and render files using linear or parallel processing and compare performance of each in the UI. The code also demonstrates a simple UI to move, rotate, and zoom the objects; to enclose objects in a bounding box; render objects in wireframe mode; display object info and stats; and allow absolute coordinates and rotations to be specified.

APPENDIX: How to Build the Project

As described earlier, the minimum requirement for developing with Vulkan APIs on Intel GPUs is a 6th Gen Intel® processor running 64-bit Windows 7, 8.1, or 10. Vulkan drivers are now included with the latest Intel HD Graphics drivers. Follow the step-by-step instructions for installing Vulkan drivers on Intel-based systems running Unity or running Unreal Engine 4, and then return here.

The following steps are for building this project using Visual Studio* 2017 from a Windows command prompt.

Preparing the build environment

1. Download the Vulkan 3D Object Viewer sample code project to a convenient folder on your hard drive.
2. Make sure your Visual Studio 2017 setup has Visual C++. If it doesn't, download and install it from https://www.visualstudio.com/downloads/.
3. The sample app relies on the Open Asset Import Library (assimp), but the pre-built version of this library doesn't work with Visual Studio 2017; it has to be re-built from scratch. Download it from http://assimp.sourceforge.net/main_downloads.html.
4. CMake is the preferred build system for assimp. You can download the latest version from https://cmake.org/ or use one from Visual Studio (YOUR_PATH_TO_MSVS\2017\Community\Common7\IDE\CommonExtensions\Microsoft\

CMake\CMake\bin). Follow these steps to build assimp:

a. Open a command prompt (cmd.exe).
b. Set “PATH=PATH_TO_CMAKE\bin;%PATH%” (skip this step If you already set this variable permanently in your system environment variables. To do that, go to: Control Panel->System->Advanced System Settings->Environment Variables and add the line above to the list).
c. Enter “cmake -f CMakeLists.txt -G "Visual Studio 15 2017 Win64".
d. Open the generated "assimp.sln" solution file in Visual Studio, go to Build->Configuration Manager and select "Release" under Configuration (unless you need to debug assimp for some reason, building the release version is recommended for the best performance).
e. Close the configuration manager and build assimp.

5. Download and install the Vulkan SDK from https://vulkan.lunarg.com/sdk/home.
6. Download and install Qt. The sample app uses Qt 5.10 UI libraries, which is the minimum version required for Vulkan support. Open-source and commercial versions will do the job here, but you'll need to register either way. To get Qt:
    a. Go to https://www.qt.io/download and select a version.
    b. Log in or register and follow prompts to set up the Qt Online Installer.
    c. Next, you'll be prompted to select a version. Pick Qt 5.10 or higher and follow prompts to install.
7. Clone or download the sample app repo to your hard drive.

Building the app

8. The file "env_setup.bat" is provided to help you set environment variables locally for the command processor. Before executing it:
    a. Open “env_setup.bat” and check whether listed variables point to the correct locations of your installed dependencies:
        i. “_VC_VARS”—path to Visual Studio environment setup vcvarsall.bat
        ii. “_QTDIR”—path to Qt root
        iii. “_VULKAN_SDK”—Vulkan SDK root
        iv. “_ASSIMP”—assimp root
        v. “_ASSIMP_BIN”—path to Release or Debug configuration of binaries
        vi. “_ASSIMP_INC”—path to assimp's header files
        vii. “_ASSIMP_LIB”—points to Release or Debug configuration of assimp lib
    b. Output from the batch file will report any paths you might have missed.
    c. Alternatively, add the following to the system's (permanent) environment variables:
        i. Create new variables:
            1. “_QTDIR”—path to Qt root
            2. “_VULKAN_SDK”—Vulkan SDK root
            3. “_ASSIMP”—assimp root
        ii. Add to variable “PATH” values:
            1. %_QTDIR%\bin
            2. %_VULKAN_SDK%\bin
            3. %_ASSIMP%\bin
        iii. Create the system variable “LIB” if it doesn't exist and add the value: %_ASSIMP%\lib
        iv. Create the system variable “INCLUDE” if it doesn't exist and add the values:

            1. %_VULKAN_SDK%\Include
            2. %_ASSIMP%\Include
        d. At the command prompt, set the current directory to the project root folder (which contains the downloaded project).
        e. Run qmake.exe.
        f. Start build:
            i. For release: nmake -f Makefile.Release
            ii. For debug: nmake -f Makefile.Debug
9. Run app:
    a. For release: WORK_DIR\release\model-viewer-using-Vulkan.exe
    b. For debug: WORK_DIR\debug\model-viewer-using-Vulkan.exe
10. Execute the newly built Vulkan object viewer app.
11. Select the number of threads to use or check "single thread." By default, the app selects the optimal number of threads based on logical cores in the host system.
12. Click "Open models..." to load some models with a selected number of threads. Then change the number of threads and click "Reload" to load the same models with new thread settings for comparison.