Project 2: Scanline algorithm

Synopsis:

Continue through the procedures of the VkApp constructor uncommenting them and following any @@ instructions until scanline functionality is achieved.

Instructions:

The following sections provide details for each step. When finished you should get the image on the right, and be able to navigate through the scene with the mouse and WASD keys.

Note: There is an error in the framework's @@ instructions. The bottom of **VkApp::myloadModel** (in **vkapp_loadModel.cpp**) has

```
//   Destroy all textures with:  for (t:m_objText) t.destroy(m_device);
//   Destroy all buffers with:   for (ob:objDesc) ob.destroy(m_device);
```

The first line is fine, but the second instruction made no sense and should be

```
//   Destroy the 4 buffers containing each object's data with:
//      for (auto& ob : m_objData) {
//         ob.vertexBuffer.destroy(m_device);
//         and similar for ob.indexBuffer, ob.matColorBuffer, ob.matIndexBuffer ... }
```

What to submit:

Submit a zip file containing

- Source code: *.cpp, *.h, all shader files.
  Do not include: Debug, **Release, .vs** or any other files maintained by Visual Studio.
- A report (in any documentation format) containing any output requested in the various @@ comments, and a screen capture of the final results, and a verification that your program runs with no validation-layer messages.
- Possibly a short video, since this **is** a realt-time application. Later projects will require a video, though it's optional for this project.

Turn on ImGui if you wish:

In vkapp.h, change

#undef GUI

to

#define GUI.

and add these lines to your **destroyAllVulkanResources** procedure

```
vkDestroyDescriptorPool(m_device, m_imguiDescPool, nullptr);
ImGui_ImplVulkan_Shutdown();
```

To test, create some GUI elements in **DrawGUI()** in **app.cpp** or temporarily uncomment the line

ImGui::ShowDemoWindow()

found there.

Preparing the post process step to display an image.

The image drawn in the first-light project was manufactured by the **postProcess** step itself. Now we modify the **postProcess** step to receive an image from wherever (the scanline pipeline in this project and the ray racing pipeline in later projects), and display it. This involves several steps:

1. *Create the scanline output buffer:*
   Uncomment and finish **createScBuffer**() (in **vkapp_scanline.cpp)**.

2. *Create a descriptor set for the post process:*
   Uncomment and finish **createPostDescriptor (**in **vkapp_scanline.cpp)**.

3. *Attach the descriptor set layout to the post process pipeline*
   Change two lines in **createPostPipeline** (in **vkapp_fns.cpp**) from
   ```
   createInfo.setLayoutCount      = 0;
   createInfo.pSetLayouts         = nullptr;
   ```
   to
   ```
   createInfo.setLayoutCount      = 1;
   createInfo.pSetLayouts         = &m_postDesc.descSetLayout;
   ```

4. *Add a variable to the **post.frag** shader to receive the image:*
   layout(set=0, binding=0) uniform sampler2D renderedImage;

5. *Bind the descriptor set by uncommenting in **postProcess:***
   //vkCmdBindDescriptorSets(m_commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
   // m_postPipelineLayout, 0, 1, &m_postDesc.descSet, 0, nullptr);

6. *Output from **post.frag**, the (slightly modified) image to the swap chain for display:*
   fragColor = pow(texture(renderedImage, uv), vec4(1.0/2.2));
   (The exponent 1/2.2 converts from linear color space to SRGB color space.)

Continue the initialization steps in VkApp::VkApp()

For scanline functionality, uncomment these calls:

```
myloadModel("models/living_room/living_room.obj", glm::mat4());
createMatrixBuffer();
createObjDescriptionBuffer();

createScanlineRenderPass();
createScDescriptorSet();
createScPipeline();
```

These procedures are implemented in **vkapp_scanline.cpp,** except for the first which is implemented in **vkapp_loadModel.cpp**. For each, find the implementation and follow any @@ instructions.

Fix for black screen. After un-commenting **myloadModel**, add these 4 lines after it.

```
app->myCamera.reset(glm::vec3(2.28, 1.68, 6.64),   0.7, -20.0,   10.66,  0.57, 0.1, 1000.0);
nonrtLightAmbient = 0.2;
nonrtLightIntensity = 1.0f;
nonrtLightPosition = vec3(0.5f, 2.5f, 3.0f);
```

The draw loop

The **drawFrame** procedure can now call the **rasterize()** function:

```
void VkApp::drawFrame()
{
    prepareFrame();

    VkCommandBufferBeginInfo beginInfo{VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO};
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vkBeginCommandBuffer(m_commandBuffer, &beginInfo);

    {   // Extra indent for code clarity
        updateCameraBuffer();

        // Draw scene
        // if (...) {
        //     raytrace();
        //     denoise();
        // }
        // else
             rasterize();

        postProcess(); //  tone mapper and output to swapchain image.

    }   // Done recording;  Execute!

    vkEndCommandBuffer(m_commandBuffer);

    submitFrame();  // Submit for display
}
```

Detailed notes for myloadModel (implemented in vkapp_loadModel.cpp)

This code uses the ASSIMP (asset import) library to read the scene into an internal representation, after which I manipulate the data into five lists:

- Vector of vertex data (1,741,337 in our case)
- Vector of indices into the vertex list, taken three at a time to define triangles. (1,741,911 indices, so 580,637 triangles)
- Vector surface material types. (44 in our case)
- Vector of material indices, one per triangle (so 580,637).
- Vector of textures (16 for our scene).

Those five lists are loaded into Vulkan buffers on the GPU, and communicated to our app through the following variables

- **vector<ObjInst> m_objInst:** Each entry pairs on object (index) and its instance transformation. Simplistically, we have exactly one object with the identity transformation.

```
struct ObjInst {
        glm::mat4 transform;    // Matrix of the instance
        uint32_t  objIndex;     // Model index
};
```

- **vector<ObjData> m_objData:** Each entry contains an objects 4 buffers

```
BufferWrap's)struct ObjData {
        ...
        BufferWrap vertexBuffer;    // Buffer of vertices
        BufferWrap indexBuffer;     // Buffer of triangle indices
        BufferWrap matColorBuffer;  // Buffer of materials
        BufferWrap matIndexBuffer;  // Buffer of each triangle's material index
};
```

- **vector<ObjDesc> m_objDesc:** Each entry contains device address of the object's buffers. This will be sent to the shader (as one entry in a descriptor set) when the object is drawn.

```
struct ObjDesc {
        ...
        uint64_t vertexAddress;         // Address of the Vertex buffer
        uint64_t indexAddress;          // Address of the index buffer
        uint64_t materialAddress;       // Address of the material buffer
        uint64_t materialIndexAddress;  // Address of the material index buffer
};
```