COMP5822M
Coursework 1

The provided solution accomplishes all five tasks listed in the coursework.
No third party libraries were used in the code, except for those provided by the coursework.

1. **Vulkan infrastructure**
   This task was completed using code from exercise 1.3, as well as its shaders. The end result of this stage was that a white triangle was rendered to a window. It is no longer visible, since it has since been replaced by the scene described by the coursework.

2. **3D Scene**
   The first step in this is to create additional shaders, since the code differentiates between textured and colored meshes. Even though there is both a texture.vert and texture.frag, only the fragment shader is required. To add them, the Makefile in cw1/cw1/shaders was modified to include the respective .spv files, which are generated in cw1/assets/shaders.

   In the project, vertex_data.hpp/cpp were added from exercise 1.4. The TextureMesh struct and its corresponding function declartions and definitions have been removed since they are not necessary. The function that creates a colorized mesh has been modified to return a vector of ColorizedMesh objects, instead of a single one and has also been made to differentiate between meshes that have texture and ones that do not. This works because the only things the ColorizedMesh struct stores is buffers. Then, the pipelines decide how to use those buffers. There are two pipelines, where the only differences are the shader paths at the start and the stride for binding 1 of the inputs, and the format for binding 1 of the vertex attributes being **sizeof(float) * 2** or **sizeof(float) * 3** and **VK_FORMAT_R32G32_SFLOAT** or **VK_FORMAT_R32G32B32_SFLOAT** for textures and colors, respectively.

   When the objects are loaded, using the tinyobj loader, each model uses the create_triangle_mesh() function in order to create their respective meshes. Then the data from the buffers is stored in vectors of buffers. This was done because there were errors, when passing the ColorizedMeshes themselves to record_commands(). Then for each texture defined, an image, image view and descriptor set is made.

   The record_commands() function is modified by putting in the relevant buffer vectors, as well as a vector of descriptor sets for each texture in the argument list. In the function itself, the draw command is placed inside a loop for the colorized meshes, then the texture pipeline is bound, then the textured mesh loop is the same as the colorized one, except that a new descriptor set is bound each iteration. This procedure renders both obj files correctly.

   Depth testing is enabled, as specified in exercise 1.4.

3. **User Camera**
   The camera was implemented with the exact controls described in the coursework specification. Various global variables were created in the cfg namespace for use of the camera. Functions were added for mouse press and position callbacks. Additional key press if statements were added to glfw_callback_key_press() for W, A, S, D, Q, E press and release. These functions flip a Boolean to true while they are held down and to false, when they are released. This is done so that movement can be continuous. The function glfw_callback_mouse_press() toggles a Boolean when the right mouse button is pressed. Finally, glfw_callback_mouse_pos() sets pitch and yaw, used for the camera, depending on the current size of the window. Window size is gotten with glfwGetWindowSize(). glfwSetInputMode() is used with GLFW_CURSOR_DISABLED so that the cursor cannot leave the window, if it is not full screen. The camera() function just sets the direction the camera looks at. It is called once before the main loop in order to set the camera matrix and render the scene and then afterwards it is called every iteration of the main loop. The function update_scene_uniforms() is updated to use glm::lookAt instead of glm::translate in order to be able to change the direction the camera is looking at. The if statements that change the position make sure to account for changes in direction. Q and E simply move the camera up and down, along the global y axis, regardless of camera direction.

4. **Mipmapping**
   In vkimage.cpp, the loop in load_image_texture2d() was changed to generate mipmaps instead of using premade ones. The contents of the loop are changed to include two additional barriers. One to transition from VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL to VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL and then another to transition it to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL for every mip level. Then, between them, vkCmdBlitImage() is used to blit the image onto itself to generate the mipmaps.

5. **Anisotropic filtering**
   In the create_device() function in vulkan_window.cpp, deviceFeatures.samplerAnisotropy is set to VK_TRUE.
   In the create_default_sampler() function in vkutill.cpp, samplerInfo.anisotropyEnable is set to VK_TRUE and samplerInfo.maxAnisotropy is set to 16.0f because that is the maximum that is allowed. The files anisotropy.png and no_anisotropy.png showcase the difference when it is enabled/disabled.

Additional note: sometimes when I build the project from scratch the first build gives an error, but compiling it again produces no errors and runs as expected. I am unsure if this is somehow a problem with my PC specifically.