

Information for the examiners

For us both it is our first time using the Vulkan API so we had to learn it from scratch.

For the base of the renderer we followed the resources provided on the site [vulkan-tutorial](#) and [Vulkan youtube series by Brendan Galea](#). Our base renderer is heavily inspired by the Vulkan youtube series from which we learnt a lot and as a result we now have a basic grasp of the vulkan API hierarchy and how the structures and classes fit together. This series was also included in the TUWEL page of the course. We felt like the tutorer created a really nice class hierarchy upon which with our little experience in Vulkan we could not improve.

- We did not end up implementing the Dynamic Environment mapping due to time limits.
- We added TinyXML library to parse the scene xml file
- For testing we used the following graphics cards:
 - Nvidia T2000
 - Apple M4

Controls

M: Toggle Camera Animation (Auto/Manual)

WASD: Move camera (Manual mode)

Arrow keys: Look around (Manual mode)

O: Toggle SSAO

C: Cycle debug mode (Normal/SSAO/Normals/Depth)

Development Status

We are using Vulkan without any framework, as we described in the Submission 1: Task Description. We also needed to include an additional library: the [stb library](#) to actually load in the textures from the hard drive. This library was not initially included in the Submission 1.

- Right now most of the core Vulkan part is implemented.
- We implemented a **render system** that users can edit the rendering pipeline configurations as they want. Right now we are using a default pipeline configured for solid, opaque, double-sided triangles with depth test enabled, and viewport size is adjustable.
- We implemented a **camera** class which adjusts the projection matrices. And it controls the camera movement along with a input management class.
- We implemented a **game object** class which has a transform component inside, and calculates the transformation calculations.
- We are using a left handed and -y up coordinate system.
- We implemented a **mesh** class that holds the vertex, index buffers also an array of Textures. A mesh object can hold multiple texture objects to support multiple textures.

- We implemented a `model` class that holds the model. The model class consists of an array of `mesh` objects. The models are dynamically passed onto the shader, only the path to the file is needed.
- We are using `assimp` as object loader.
- We can assign textures to objects, and it can be rendered.
- We implemented a `descriptor` class that generates and holds the descriptor pool and also the descriptor set layout and the descriptor set. Only support for a single descriptor set is implemented currently.
- We implemented a `lightmanager` class that handles the position and intensity values of a number of `point` lights and one `directional` light.
- We implemented `diffuse shading` for now.
- No complex effect is implemented yet.

What We Have Done After Assignment 2

- We implemented 3 complex effects:
 - SSAO
 - Compute Shader based Particle System
 - Tangent-Space Normal Mapping

SSAO

- For SSAO, we changed our rendering pipeline from forward rendering pipeline to a deferred rendering pipeline.
- We have 4 passes in our deferred rendering pipeline:
 1. G-Buffer Pass
 2. SSAO Pass
 3. SSAO Blur Pass
 4. Lighting Pass
 - G-Buffer texture is calculated at the first pass in `gbuffer` shaders
 - SSAO is calculated with gbuffer texture in `ssao` shaders.
 - A hemisphere kernel is used to make flat surfaces affected by SSAO.
 - Blur is applied on `ssao_blur` shader.
 - And final lighting is applied on `triangle` shader.
- SSAO can be toggled by pressing O. And you can realize the SSAO effect by looking to the car or the viking room.

Compute shader based Particle System

- I first implemented a compute shader, which in 256 batches updates the particles based on their velocities and their direction. They start off as dark gray and as their ttl(Time To Live) lowers they whiten out, becoming fully white before disappearing. When the ttl of a particle reaches 0 it is reset to the point of the parent and then it restarts its journey.

- Implementing it vulkan side was really challenging and I ended up creating a new pipeline where a graphics shader displays the results of the compute. The particles are sent to the gpu at start up and then they live in the gpu until the program is terminated, after which they are properly destroyed.
- In the demo program we are simulating ~132.000 particles, more than required.
- This effect can be seen as the smoke coming out of the bonnet of the car

Tangent-Space Normal Mapping

- Assimp has a handy flag that generates the tangent vector when importing a mesh. I save these tangents and then send them per vertex to the vertex shader. Here from the cross product of the normal and the tangent we can calculate the bitangent. With these three we have the TBN matrix with which we can transform the normaltextures to the proper space. This is done inside the fragment shader
- We saw the result of this best in the road, the small bumps and cracks are now visible on it, while before they were not.