

# Project Documentation

---

## Brief description of the implementation

This project is a 3D rendering application developed in C++ using the Vulkan graphics API. It utilizes the `vcpkg` package manager for dependency management. The application sets up a Vulkan-based rendering pipeline to display complex 3D scenes. For assignment 3, it loads a city scene and plays a short animated scene involving an extraterrestrial blob.

## Scene Loading

The scene is loaded from a glTF 2.0 file using the `fastgltf` library. The loading process involves the following steps:

1. **Parsing:** The glTF file is parsed to extract scene data, including meshes, materials, textures, and lights.
2. **Mesh Data:** Vertex data (positions, normals, tangents, texture coordinates) and index data are extracted and stored in buffers. Bounding boxes are also calculated for each mesh primitive for culling purposes.
3. **Image Data:** Images are loaded from buffers using the `stb_image` library.
4. **Material Data:** Materials are processed, and their properties (albedo, metalness, roughness, etc.) are extracted. The loader also handles texture mapping and can merge occlusion, roughness, and metallic textures into a single ORM texture.
5. **Animation Data:** Animations channels (translation, rotation, scaling) are loaded and later applied to instances with meshes, spot-lights and point-lights, and the "Camera" and "Blob" nodes.
6. **Light Data:** Punctual lights (directional, point, and spot) are loaded.
7. **Node Hierarchy:** The scene graph is traversed to process nodes and their transformations.

The loaded data is organized into a `gltf::Scene` object, which is then used by the rendering system.

## Rendering

The rendering process is managed by the `RenderSystem` class and is divided into several passes:

1. **Depth Pre-Pass:** The `DepthPrePassRenderer` renders the scene into the main depth buffer. Frustum culling is used to avoid rendering objects that are outside the camera's view.

2. **SSAO Pass:** The `SSAORenderer` generates screen-space ambient occlusion based on the scene depth. This pass can be executed on the compute queue (optionally async) and writes into AO intermediary/result images that are later consumed by the main scene pass.
3. **Light Pass:** The `LightRenderer` builds per-tile light lists for the visible lights in the scene. This pass is executed as a compute pass and writes light indices into a storage buffer that is used during shading.
4. **Shadow Pass:** The `ShadowRenderer` generates shadow maps for the main light source. This is done by rendering the scene from the light's point of view into cascaded depth buffers.
5. **Blob Compute Pass:** The `BlobRenderer` computes the metaball object.
6. **Main Render Pass:** The `PbrSceneRenderer` renders the main scene using a physically-based rendering (PBR) approach. It uses the previously generated shadow maps to apply shadows to the scene. Frustum culling is used to avoid rendering objects that are outside the camera's view. For optimal performance we use GPU-driven rendering. By using bindless descriptors we can render the whole scene with a single draw-indirect call.
7. **Skybox Render Pass:** The `SkyboxRenderer` draws a skybox to provide a background for the scene.
8. **Blob Render Pass:** The `BlobRenderer` computes and renders the metaball object.
9. **Fog Pass:** The `FogLightRenderer` builds clustered light lists for volumetric fog and the `FogRenderer` renders volumetric fog into the scene.
10. **Bloom Pass:** The `BloomRenderer` generates bloom from the HDR scene color and outputs a bloom texture that is composited during final post-processing.
11. **Post-Processing Pass:** The `FinalizeRenderer` applies post-processing effects, such as tone mapping, to the rendered image.
12. **ImGui Pass:** The `ImGuiBackend` renders the user interface, which can be used to control various rendering settings.

The `RenderSystem` uses a frame-in-flight synchronization mechanism to manage the rendering and presentation of frames efficiently.

## Used Libraries

The following additional libraries were used in this project:

- **glfw3:** A multi-platform library for OpenGL, OpenGL ES, Vulkan, window and input. (<https://www.glfw.org/>)
- **glm:** A header-only C++ mathematics library for graphics software. (<https://glm.g-truc.net/>)
- **glslang:** A shader front end and validator. (<https://github.com/KhronosGroup/glslang>)

- **shaderc**: A collection of tools, libraries, and tests for shader compilation. (<https://github.com/google/shaderc>)
- **stb**: A collection of single-file public domain libraries for C/C++. (<https://github.com/nothings/stb>)
- **Vulkan**: A new generation graphics and compute API that provides high-efficiency, cross-platform access to modern GPUs. (<https://www.vulkan.org/>)
- **imgui**: A bloat-free graphical user interface library for C++. (<https://github.com/ocornut/imgui>)
- **vk-bootstrap**: A library to help with Vulkan boilerplate setup. (<https://github.com/charles-lunarg/vk-bootstrap>)
- **fastgltf**: A fast glTF 2.0 parser. (<https://github.com/spnda/fastgltf>)
- **Vulkan Memory Allocator**: A library to help with Vulkan memory allocation. (<https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator>)

## Effects

### Complex Effects

- **Shadow Map with manual PCF & Cascaded Shadow Maps**: We have five shadow cascades, each has a x2048 resolution. The shadow map draw uses frustum culling and objects entirely contained in a lower level cascade are not rendered again in the higher levels. Shadow volumes are tightly fitted to the view frustum splits. During the main render pass we find the lowest cascade level for every fragment, allowing for maximum fidelity. Additionally, a 9-tap poisson disk sampling is used.
- **Forward+ (tiled forward rendering)** We dispatch a compute shader invocation for every pixel. Using a local group size of 16x16, the screen is split into tiles. Each group calculates the tile depth bounds from the depth buffer using fast subgroup instructions and shared variables. The min and max depths are used as the near and far bounds of the tile frustum. Checking frustum or AABB vs Cone (Spotlight) intersection is computationally difficult. Instead, we calculate a tapered capsule (aka. beam) bounding geometry for our frustum. This novel idea allows for an exact cone and sphere intersection test. Finally, every group does a strided loop over all lights and appends the intersecting light indices into a segmented buffer. The buffer is segmented such that every tile can have 256-1 lights assigned. You can view the light density per tile by enabling "Light Density" under the "Rendering" tab.
- **Volumetric Lighting**: Similar to our light-tile assignment compute shader, for volumetric rendering we assign lights to view frustum "froxels". Here we ignore depth bounds because relevant lights can be floating in mid air. Intersection is again done using the tapered capsule (aka beam) intersection. We use 32x18x24 (WxHxD) frustum voxels, with each having a maximum of 128 lights. The fog is rendered at half resolution and then upsampled using a depth aware filter, similar to our AO.

- **Ambient Occlusion:** We implemented a modified version of GTAO (Ground Truth Ambient Occlusion). We don't do the depth pre-processing (calculating edges and LODs). This reduces performance for large view-space radii, but simplifies the implementation a lot. Also, the reference implementation is broken and produces unwanted darkening at the edges of the screen, this is fixed in our implementation. The SSAO is rendered at half resolution and upscaled using a depth-aware bilateral blur filter. We found that 3 slices with 6 samples per slice direction give good results.
- **Blobby Object using Marching Cubes:** Dynamic mesh generated during runtime based on an SDF and the marching cubes algorithm.

## Simple effects

- **Bloom:** Implemented following the "Deferred Rendering in Killzone 2" presentation and the "Physically Based Bloom" Learn OpenGL article.
- **Frustum Culling:** To review this effect, you can turn on the debug menu (F3) and enable "Pause Culling" under the "Rendering" tab.
- **Physically Based Shading**
- **Hierarchical Animation**

## Code Reusage from Other Projects

- The **input system**, **audio system** and the **bloom effect** were mostly copied from the PTVC project.
- The **shadow** and **ambient occlusion rendering** are loosely based on an implementation for the PTVC project but were modified and improved heavily.

## Testing Environment

The project was tested on the following graphics cards:

- NVIDIA RTX 5070 Ti
- NVIDIA RTX 2060 Super

## Controls

- F1 - Toggle Debug Camera
- F3 - Show Debug GUI
- W/A/S/D - Strafe

- Space/Ctrl - Up and Down
- Shift - Go Fast
- Alt/Esc - Release Mouse
- Mouse - Look Around