VulkanRenderer

Project: AcceleRender
Authors: Finley Deevy, Eric Newton
Version: 1.0
Date: 2025-09-02

---

1. Overview

The `VulkanRenderer` class encapsulates a complete Vulkan 1.3 rendering engine using C++ RAII.
It manages the full lifecycle of Vulkan rendering, including:

- Creating a window using GLFW
- Initializing Vulkan (instance, device, surface, queues)
- Creating and managing the swap chain
- Creating the graphics pipeline
- Managing buffers (vertex, index, uniform)
- Texture image creation and mipmap generation
- Depth buffering and multisample anti-aliasing (MSAA)
- Command buffer allocation and recording
- Synchronization between CPU and GPU
- Rendering frames in a loop

The class uses `vk::raii` wrapper objects, meaning that when an object goes out of scope, Vulkan resources are destroyed automatically without needing manual cleanup.

---

2. Design Goals & Principles

1. **RAII Resource Ownership**
   Vulkan handles are wrapped in RAII containers so cleanup is automatic.

2. **Explicit GPU Control**
   Functions expose Vulkan operations clearly (buffer creation, memory allocation, layout transitions).
3. **Single Responsibility Structure**
   Each setup action has a dedicated method, making initialization steps predictable.
4. **Stable Rendering Loop**
   Uses a multi-frame-in-flight system with semaphores and fences to keep CPU and GPU synchronized efficiently.
5. **Swap Chain Resilience**
   Handles window resizing and swap chain invalidation robustly.

---

3. Architecture Summary

The class follows this lifecycle:

1. Initialize the window (GLFW)
2. Initialize Vulkan
3. Enter the render loop (`run() / mainLoop()`)
4. Clean up Vulkan and window resources

Initialization includes:

- Creating instance, surface, debug messenger
- Selecting a physical GPU and creating a logical device
- Creating swap chain, pipeline, descriptor sets, buffers, images

Rendering includes:

- Acquiring an image from the swap chain
- Recording commands into a command buffer
- Submitting commands to the GPU queue
- Presenting the rendered image

Cleanup is automatic for everything stored in a `vk::raii` object.

---

4. Key Subsystems

4.1 Window and Vulkan Context Initialization

Responsible methods:

- `initWindow()`
- `createInstance()`
- `setupDebugMessenger()`
- `createSurface()`
- `pickPhysicalGPU()`
- `pickLogicalGPU()`

This creates the application window and prepares Vulkan for rendering.

4.2 Swap Chain Management

Responsible methods:

- `createSwapChain()`
- `chooseSwapSurfaceFormat()`
- `chooseSwapPresentMode()`
- `chooseSwapExtent()`
- `recreateSwapChain()`
- `cleanupSwapChain()`

The swap chain determines how rendered images are presented.
`recreateSwapChain()` is called whenever the window is resized.

4.3 Graphics Pipeline

Responsible methods:

- `createGraphicsPipeline()`
- `createDescriptorSetLayout()`
- `createShaderModule()`

Configures shader stages, rasterizer, blending, viewport/scissor, and depth settings.

4.4 GPU Resources

Responsible methods include:

- Buffers:
  - `createBuffer()`
  - `copyBuffer()`
  - `createVertexBuffer()`
  - `createIndexBuffer()`
- Textures:
  - `createTextureImage()`
  - `createTextureImageView()`
  - `createTextureSampler()`
  - `generateMipmaps()`
- Images:
  - `createDepthResources()`
  - `createColorResources()`
  - `transitionImageLayout()`
  - `copyBufferToImage()`

These handle GPU memory allocation and uploading data from CPU to GPU.

4.5 Command Recording and Rendering Loop

- `createCommandPool()`
- `createCommandBuffers()`
- `recordCommandBuffer()`
- `drawFrame()`

Each frame:

1. Acquire swap chain image
2. Update uniform buffer
3. Execute command buffer
4. Present result to screen

4.6 Synchronization System

- `createSyncObjects()`
- Uses semaphores and fences:
  - One to wait for image availability
  - One to signal frame render completion

The renderer supports multiple frames in flight.

---

5. Important Class Attributes

The class stores:

- Vulkan handles (instance, device, surface, queues)
- Swap chain resources (swap chain object, image views, image format, extent)
- Pipeline resources (pipeline layout, graphics pipeline)
- Buffers (vertex, index, uniform)
- Images (texture, depth, MSAA color attachment)
- Synchronization primitives (fences, semaphores)
- CPU-side vertex and index data loaded from models
- Per-frame state (current frame index, framebuffer resized flag)

---

6. Debugging and Validation

- Validation layers are enabled during development
- A debug callback logs messages and Vulkan warnings
- Functions validate GPU capabilities (depth formats, memory type availability, MSAA sample count)

---

7. Future Improvements

- Multiple model support (dynamic scene graph)
- Live shader reloading
- GUI overlay with ImGui
- GPU memory pooling for large scene workloads

---

8. Limitations

- Only supports one window at a time
- Pipelines must be recreated when swap chain changes
- No hot-reload for shaders (yet)

---

9. Conclusion

`VulkanRenderer` is designed to provide a structured, RAII-managed Vulkan rendering engine. The design emphasizes clarity, safety, separation of responsibilities, and clean teardown. It forms a solid foundation for more complex rendering features.