**Minecarft jarock**

package com.jarock.engine;﻿import com.jarock.nativecore.\*;﻿import com.jarock.modapi.registry.Registry;﻿import com.jarock.modapi.Spell;﻿import com.jarock.samplemods.\*;﻿import com.jarock.world.WorldGen;﻿import com.[jarock.net](jarock.net#jarock.net).UdpServer;﻿import com.[jarock.net](jarock.net#jarock.net).UdpClient;﻿import com.jarock.ecs.\*;﻿import com.jarock.engine.[cmd.Com](cmd.Com#cmd.Com)mandRouter;﻿import java.nio.\*;﻿import java.util.\*;﻿public final class Engine {﻿    private static final Registry<Spell> SPELLS = new Registry<>();﻿    private static final World WORLD = new World();﻿    private static final List<ItemStack> ITEMS = new ArrayList<>();﻿    private static final SharedState SHARED\_STATE = new SharedState(1000); // Max 1000 entities﻿    private static final int CHUNK\_SIZE = 16;﻿    private static final Map<Integer, Long> chunkMeshes = new HashMap<>();﻿    private static final long TICK\_NS = 50\_000\_000L; // 50ms/tick (20 tick/s)﻿    public static void main(String[] args) throws Exception {﻿        NativeCore.nRendererInit(0, 1280, 720); // Vulkan, 720p﻿        NativeCore.nSetSharedBuffer(SHARED\_STATE.getBuffer(), SHARED\_STATE.getMaxEntities());﻿        RustCore.initSharedBuffer(SHARED\_STATE.getBuffer(), SHARED\_STATE.getMaxEntities());﻿        // Register spells﻿        SPELLS.register("jarock:holy\_light", new HolyLightSpell());﻿        SPELLS.register("jarock:dark\_binding", new DarkBindingSpell());﻿        // Register commands﻿        CommandRouter router = new CommandRouter();﻿        router.reg("give\_adrenaline", (a, ctx) -> {﻿            ITEMS.add(new ItemStack(new AdrenalineSyringe(), 1, 0, 100, 0));﻿            System.out.println("[OP] Gave Adrenaline Syringe");﻿        });﻿        router.reg("cast", (a, ctx) -> {﻿            var id = a[0];﻿            double x = Double.parseDouble(a[1]);﻿            double y = Double.parseDouble(a[2]);﻿            double z = Double.parseDouble(a[3]);﻿            var sp = SPELLS.get(id);﻿            if (sp != null) sp.cast(x, y, z);﻿            else System.out.println("No spell: " + id);﻿        });﻿        router.reg("host\_lan", (a, ctx) -> {﻿            int port = (a.length > 0 ? Integer.parseInt(a[0]) : 25565);﻿            new UdpServer(port);﻿            System.out.println("Hosting LAN on " + port);﻿        });﻿        router.reg("join\_lan", (a, ctx) -> {﻿            String host = a[0];﻿            int port = Integer.parseInt(a[1]);﻿            new UdpClient(host, port);﻿            System.out.println("Joining LAN at " + host + ":" + port);﻿        });﻿        router.reg("break\_block", (a, ctx) -> {﻿            int x = Integer.parseInt(a[0]), y = Integer.parseInt(a[1]), z = Integer.parseInt(a[2]);﻿            WORLD.setBlock(x, y, z, (byte)0);﻿            ITEMS.add(new ItemStack(new AdrenalineSyringe(), 1, x, y, z));﻿            System.out.println("Broke block at (" + x + "," + y + "," + z + ")");﻿        });﻿        // Generate world﻿        WorldGen worldGen = new WorldGen(1234);﻿        ByteBuffer voxels = ByteBuffer.allocateDirect(CHUNK\_SIZE \* 256 \* CHUNK\_SIZE);﻿        ByteBuffer meshOut = ByteBuffer.allocateDirect(4 \* 1024 \* 1024);﻿        ByteBuffer idx = ByteBuffer.allocateDirect(1024 \* 1024);﻿        ByteBuffer instances = ByteBuffer.allocateDirect(1024 \* 6 \* Float.BYTES);﻿        FloatBuffer viewProj = FloatBuffer.allocate(16);﻿        viewProj.put(new float[]{﻿            1, 0, 0, 0,﻿            0, 1, 0, 0,﻿            0, 0, 1, 0,﻿            0, 0, 0, 1﻿        }).flip();﻿        // Generate chunks﻿        for (int cx = -2; cx <= 2; cx++) for (int cz = -2; cz <= 2; cz++) {﻿            worldGen.fillChunkVoxels(cx, cz, voxels);﻿            long packed = NativeCore.nBuildChunkMesh(voxels, CHUNK\_SIZE, 256, CHUNK\_SIZE, meshOut);﻿            int vertices = (int)(packed & 0xffffffffL);﻿            long bytes = (packed >>> 32);﻿            long meshHandle = NativeCore.nCreateMesh(meshOut, vertices, idx, (int)(bytes - vertices \* 6 \* Float.BYTES) / Short.BYTES);﻿            chunkMeshes.put((cx << 16) | cz, meshHandle);﻿            instances.putFloat(cx \* CHUNK\_SIZE).putFloat(0).putFloat(cz \* CHUNK\_SIZE);﻿            instances.putFloat(1).putFloat(1).putFloat(1);﻿        }﻿        instances.flip();﻿        // Spawn 50 dragons﻿        for (int i = 0; i < 50; i++) {﻿            World.EID dragon = WORLD.spawn();﻿            new MiniOceanDragon().onSpawn(WORLD, dragon);﻿        }﻿        // Game loop (fixed timestep)﻿        long lastTime = System.nanoTime();﻿        long accumulator = 0;﻿        float playerX = 0, playerY = 100, playerZ = 0; // Fake player pos﻿        while (true) {﻿            long now = System.nanoTime();﻿            accumulator += now - lastTime;﻿            lastTime = now;﻿            // Logic update (20 tick/s)﻿            while (accumulator >= TICK\_NS) {﻿                List<World.EID> nearby = WORLD.getNearby(playerX, playerY, playerZ, 32.0f);﻿                for (World.EID eid : nearby) {﻿                    Transform t = WORLD.get(eid, Transform.class);﻿                    Velocity v = WORLD.get(eid, Velocity.class);﻿                    AIThink ai = WORLD.get(eid, AIThink.class);﻿                    if (t != null && v != null) {﻿                        Transform newT = new Transform(t.x() + v.vx() \* 0.05f, t.y() + v.vy() \* 0.05f, t.z() + v.vz() \* 0.05f);﻿                        WORLD.updateTransform(eid, t, newT);﻿                        SHARED\_STATE.updateEntity(eid.id(), newT.x(), newT.y(), newT.z());﻿                    }﻿                    if (ai != null && ai.behavior().equals("dragon")) {﻿                        RustCore.updateDragon(eid.id(), playerX, playerZ);﻿                    }﻿                }﻿                // Update items﻿                for (int i = 0; i < ITEMS.size(); i++) {﻿                    ItemStack stack = ITEMS.get(i);﻿                    float dist = (float)Math.sqrt(Math.pow(stack.pos().x() - playerX, 2) + Math.pow(stack.pos().z() - playerZ, 2));﻿                    if (dist > 32.0f) continue;﻿                    for (int j = i + 1; j < ITEMS.size(); j++) {﻿                        ItemStack other = ITEMS.get(j);﻿                        if (Math.abs(stack.pos().x() - other.pos().x()) < 1.0f &&﻿                            Math.abs(stack.pos().y() - other.pos().y()) < 1.0f &&﻿                            Math.abs(stack.pos().z() - other.pos().z()) < 1.0f) {﻿                            if ([stack.me](stack.me#stack.me)rge(other)) {﻿                                if (other.count() == 0) ITEMS.remove(j--);﻿                            }﻿                        }﻿                    }﻿                    SHARED\_STATE.updateEntity(1000 + i, stack.pos().x(), stack.pos().y(), stack.pos().z());﻿                }﻿                accumulator -= TICK\_NS;﻿            }﻿            // Render﻿            for (var e : chunkMeshes.entrySet()) {﻿                NativeCore.nSubmitMesh(e.getValue(), instances, instances.remaining() / (6 \* Float.BYTES), viewProj);﻿            }﻿            // Fake player input﻿            playerX += 0.1f; playerZ += 0.1f;﻿            SHARED\_STATE.updateEntity(0, playerX, playerY, playerZ);﻿            Thread.sleep(16);﻿        }﻿    }﻿}﻿mod logic;﻿mod shared;﻿#[no\_mangle]﻿pub extern "C" fn init() {﻿    shared::init();﻿}﻿use std::slice;﻿static mut SHARED\_BUFFER: \*mut u8 = std::ptr::null\_mut();﻿static mut MAX\_ENTITIES: usize = 0;﻿pub fn init() {}﻿#[no\_mangle]﻿pub extern "C" fn init\_shared\_buffer(ptr: \*mut u8, max\_entities: usize) {﻿    unsafe {﻿        SHARED\_BUFFER = ptr;﻿        MAX\_ENTITIES = max\_entities;﻿    }﻿}﻿#[no\_mangle]﻿pub extern "C" fn update\_dragon(eid: i32, player\_x: f32, player\_z: f32) {﻿    unsafe {﻿        let offset = eid as usize \* 16;﻿        if offset + 16 > MAX\_ENTITIES \* 16 { return; }﻿        let buf = slice::from\_raw\_parts\_mut(SHARED\_BUFFER, MAX\_ENTITIES \* 16);﻿        let x = f32::from\_le\_bytes([buf[offset + 4], buf[offset + 5], buf[offset + 6], buf[offset + 7]]);﻿        let z = f32::from\_le\_bytes([buf[offset + 12], buf[offset + 13], buf[offset + 14], buf[offset + 15]]);﻿        let dx = player\_x - x;﻿        let dz = player\_z - z;﻿        let len = (dx \* dx + dz \* dz).sqrt();﻿        if len > 0.1 {﻿            let vx = dx / len \* 0.2;﻿            let vz = dz / len \* 0.2;﻿            let new\_x = x + vx \* 0.05;﻿            let new\_z = z + vz \* 0.05;﻿            buf[offset + 4..offset + 8].copy\_from\_slice(&new\_x.to\_le\_bytes());﻿            buf[offset + 12..offset + 16].copy\_from\_slice(&new\_z.to\_le\_bytes());﻿        }﻿    }﻿}﻿#pragma once﻿#include <cstdint>﻿#include <cstddef>﻿namespace jarock {﻿struct MeshBuildStats {﻿    uint32\_t vertices;﻿    uint32\_t indices;﻿};﻿size\_t build\_chunk\_mesh(const uint8\_t\* voxels, int sx, int sy, int sz,﻿                        uint8\_t\* outBuf, size\_t outCap, MeshBuildStats\* stats) noexcept;﻿void renderer\_init(void\* nwh, uint32\_t width, uint32\_t height) noexcept;﻿void renderer\_shutdown() noexcept;﻿void set\_shared\_buffer(uint8\_t\* ptr, size\_t maxEntities) noexcept;﻿uint64\_t create\_mesh(const float\* verts, uint32\_t vcount, const uint16\_t\* idx, uint32\_t icount) noexcept;﻿void submit\_mesh(uint64\_t pack, const float\* instances, uint32\_t instanceCount, const float\* viewProj) noexcept;﻿}﻿#include "jarock\_renderer.h"﻿#include <vulkan/vulkan.h>﻿#include <vector>﻿#include <stdexcept>﻿static VkInstance instance;﻿static VkPhysicalDevice physicalDevice;﻿static VkDevice device;﻿static VkQueue graphicsQueue;﻿static VkQueue presentQueue;﻿static VkSwapchainKHR swapchain;﻿static std::vector<VkImage> swapchainImages;﻿static VkRenderPass renderPass;﻿static VkPipeline pipeline;﻿static VkDescriptorSetLayout descriptorSetLayout;﻿static VkDescriptorPool descriptorPool;﻿static std::vector<VkDescriptorSet> descriptorSets;﻿static VkImage textureImage;﻿static VkDeviceMemory textureImageMemory;﻿static VkImageView textureImageView;﻿static VkSampler textureSampler;﻿static uint8\_t\* sharedBuffer = nullptr;﻿static size\_t maxEntities = 0;﻿namespace jarock {﻿void renderer\_init(void\* nwh, uint32\_t width, uint32\_t height) noexcept {﻿    try {﻿        VkApplicationInfo appInfo{};﻿        appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;﻿        appInfo.pApplicationName = "Jarock";﻿        appInfo.applicationVersion = VK\_MAKE\_VERSION(1, 0, 0);﻿        appInfo.pEngineName = "Jarock Engine";﻿        appInfo.engineVersion = VK\_MAKE\_VERSION(1, 0, 0);﻿        appInfo.apiVersion = VK\_API\_VERSION\_1\_0;﻿        VkInstanceCreateInfo createInfo{};﻿        createInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;﻿        createInfo.pApplicationInfo = &appInfo;﻿        // Add platform-specific extensions (Android Surface for OPPO)﻿        const char\* extensions[] = { "VK\_KHR\_surface", "VK\_KHR\_android\_surface" };﻿        createInfo.enabledExtensionCount = 2;﻿        createInfo.ppEnabledExtensionNames = extensions;﻿        if (vkCreateInstance(&createInfo, nullptr, &instance) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create Vulkan instance");﻿        }﻿        // Placeholder: Initialize physical device, device, queues, swapchain, render pass, pipeline, texture﻿        // (Simplified for brevity, full Vulkan init would be ~500 lines)﻿    } catch (const std::exception& e) {﻿        printf("Renderer init failed: %s\n", e.what());﻿    }﻿}﻿void renderer\_shutdown() noexcept {﻿    vkDestroyPipeline(device, pipeline, nullptr);﻿    vkDestroyRenderPass(device, renderPass, nullptr);﻿    vkDestroySwapchainKHR(device, swapchain, nullptr);﻿    vkDestroyDescriptorPool(device, descriptorPool, nullptr);﻿    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);﻿    vkDestroySampler(device, textureSampler, nullptr);﻿    vkDestroyImageView(device, textureImageView, nullptr);﻿    vkDestroyImage(device, textureImage, nullptr);﻿    vkFreeMemory(device, textureImageMemory, nullptr);﻿    vkDestroyDevice(device, nullptr);﻿    vkDestroyInstance(instance, nullptr);﻿}﻿void set\_shared\_buffer(uint8\_t\* ptr, size\_t maxEntities) noexcept {﻿    sharedBuffer = ptr;﻿    jarock::maxEntities = maxEntities;﻿}﻿uint64\_t create\_mesh(const float\* verts, uint32\_t vcount, const uint16\_t\* idx, uint32\_t icount) noexcept {﻿    // Placeholder: Create Vulkan vertex/index buffers﻿    return 0;﻿}﻿void submit\_mesh(uint64\_t pack, const float\* instances, uint32\_t instanceCount, const float\* viewProj) noexcept {﻿    try {﻿        // Read shared buffer for entity positions﻿        std::vector<float> entityInstances;﻿        for (size\_t i = 0; i < maxEntities; i++) {﻿            uint32\_t id = \*(uint32\_t\*)(sharedBuffer + i \* 16);﻿            float x = \*(float\*)(sharedBuffer + i \* 16 + 4);﻿            float y = \*(float\*)(sharedBuffer + i \* 16 + 8);﻿            float z = \*(float\*)(sharedBuffer + i \* 16 + 12);﻿            entityInstances.push\_back(x); entityInstances.push\_back(y); entityInstances.push\_back(z);﻿            entityInstances.push\_back(1); entityInstances.push\_back(1); entityInstances.push\_back(1); // Color﻿        }﻿        // Placeholder: Vulkan draw call﻿    } catch (const std::exception& e) {﻿        printf("Render failed: %s\n", e.what());﻿    }﻿}﻿}﻿pkg install clang cmake ninja git openjdk-17 rust﻿termux-setup-storage﻿cd /sdcard/jarock/native﻿mkdir build && cd build﻿cmake -G Ninja .. -DCMAKE\_TOOLCHAIN\_FILE=$ANDROID\_NDK/build/cmake/android.toolchain.cmake -DANDROID\_ABI=arm64-v8a﻿ninja﻿cp libjarock\_core.so ../../java/lib/﻿cd /sdcard/jarock/rust﻿cargo build --release --target aarch64-linux-android﻿cp target/aarch64-linux-android/release/libjarock\_rust.so ../java/lib/﻿java -Djava.library.path=lib -jar build/libs/java.jar﻿#pragma once﻿#include <vulkan/vulkan.h>﻿#include <cstdint>﻿namespace jarock {﻿struct Mesh {﻿    VkBuffer vertexBuffer;﻿    VkDeviceMemory vertexMemory;﻿    VkBuffer indexBuffer;﻿    VkDeviceMemory indexMemory;﻿    uint32\_t vertexCount;﻿    uint32\_t indexCount;﻿};﻿void renderer\_init(void\* nwh, uint32\_t width, uint32\_t height) noexcept;﻿void renderer\_shutdown() noexcept;﻿void set\_shared\_buffer(uint8\_t\* ptr, size\_t maxEntities) noexcept;﻿uint64\_t create\_mesh(const float\* verts, uint32\_t vcount, const uint16\_t\* idx, uint32\_t icount) noexcept;﻿void submit\_mesh(uint64\_t pack, const float\* instances, uint32\_t instanceCount, const float\* viewProj) noexcept;﻿}﻿#include "jarock\_renderer.h"﻿#include <stdexcept>﻿#include <vector>﻿#include <cstring>﻿#include <android/native\_window.h>﻿static VkInstance instance;﻿static VkPhysicalDevice physicalDevice;﻿static VkDevice device;﻿static VkQueue graphicsQueue;﻿static VkQueue presentQueue;﻿static VkSwapchainKHR swapchain;﻿static std::vector<VkImage> swapchainImages;﻿static std::vector<VkImageView> swapchainImageViews;﻿static VkRenderPass renderPass;﻿static VkPipelineLayout pipelineLayout;﻿static VkPipeline pipeline;﻿static VkDescriptorSetLayout descriptorSetLayout;﻿static VkDescriptorPool descriptorPool;﻿static std::vector<VkDescriptorSet> descriptorSets;﻿static VkImage textureImage;﻿static VkDeviceMemory textureImageMemory;﻿static VkImageView textureImageView;﻿static VkSampler textureSampler;﻿static VkCommandPool commandPool;﻿static std::vector<VkCommandBuffer> commandBuffers;﻿static std::vector<VkFramebuffer> framebuffers;﻿static uint8\_t\* sharedBuffer = nullptr;﻿static size\_t maxEntities = 0;﻿namespace jarock {﻿static void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties,﻿                         VkBuffer& buffer, VkDeviceMemory& memory) {﻿    VkBufferCreateInfo bufferInfo{};﻿    bufferInfo.sType = VK\_STRUCTURE\_TYPE\_BUFFER\_CREATE\_INFO;﻿    bufferInfo.size = size;﻿    bufferInfo.usage = usage;﻿    bufferInfo.sharingMode = VK\_SHARING\_MODE\_EXCLUSIVE;﻿    if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK\_SUCCESS) {﻿        throw std::runtime\_error("Failed to create buffer");﻿    }﻿    VkMemoryRequirements memReq;﻿    vkGetBufferMemoryRequirements(device, buffer, &memReq);﻿    VkPhysicalDeviceMemoryProperties memProps;﻿    vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProps);﻿    uint32\_t memTypeIdx = -1;﻿    for (uint32\_t i = 0; i < [memProps.me](memProps.me#memProps.me)moryTypeCount; i++) {﻿        if (([memReq.me](memReq.me#memReq.me)moryTypeBits & (1 << i)) && ([memProps.me](memProps.me#memProps.me)moryTypes[i].propertyFlags & properties) == properties) {﻿            memTypeIdx = i;﻿            break;﻿        }﻿    }﻿    if (memTypeIdx == -1) throw std::runtime\_error("Failed to find memory type");﻿    VkMemoryAllocateInfo allocInfo{};﻿    allocInfo.sType = VK\_STRUCTURE\_TYPE\_MEMORY\_ALLOCATE\_INFO;﻿    allocInfo.allocationSize = memReq.size;﻿[allocInfo.me](allocInfo.me#allocInfo.me)moryTypeIndex = memTypeIdx;﻿    if (vkAllocateMemory(device, &allocInfo, nullptr, &memory) != VK\_SUCCESS) {﻿        throw std::runtime\_error("Failed to allocate memory");﻿    }﻿    vkBindBufferMemory(device, buffer, memory, 0);﻿}﻿void renderer\_init(void\* nwh, uint32\_t width, uint32\_t height) noexcept {﻿    try {﻿        // Create Vulkan instance﻿        VkApplicationInfo appInfo{};﻿        appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;﻿        appInfo.pApplicationName = "Jarock";﻿        appInfo.applicationVersion = VK\_MAKE\_VERSION(1, 0, 0);﻿        appInfo.pEngineName = "Jarock Engine";﻿        appInfo.engineVersion = VK\_MAKE\_VERSION(1, 0, 0);﻿        appInfo.apiVersion = VK\_API\_VERSION\_1\_0;﻿        VkInstanceCreateInfo createInfo{};﻿        createInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;﻿        createInfo.pApplicationInfo = &appInfo;﻿        const char\* extensions[] = { "VK\_KHR\_surface", "VK\_KHR\_android\_surface" };﻿        createInfo.enabledExtensionCount = 2;﻿        createInfo.ppEnabledExtensionNames = extensions;﻿        if (vkCreateInstance(&createInfo, nullptr, &instance) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create Vulkan instance");﻿        }﻿        // Pick physical device﻿        uint32\_t deviceCount = 0;﻿        vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);﻿        std::vector<VkPhysicalDevice> devices(deviceCount);﻿        vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());﻿        physicalDevice = devices[0]; // Simplified﻿        // Create device and queues﻿        VkDeviceQueueCreateInfo queueInfo{};﻿        queueInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_QUEUE\_CREATE\_INFO;﻿        queueInfo.queueFamilyIndex = 0; // Simplified﻿        queueInfo.queueCount = 1;﻿        float queuePriority = 1.0f;﻿        queueInfo.pQueuePriorities = &queuePriority;﻿        VkDeviceCreateInfo deviceInfo{};﻿        deviceInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_CREATE\_INFO;﻿        deviceInfo.queueCreateInfoCount = 1;﻿        deviceInfo.pQueueCreateInfos = &queueInfo;﻿        const char\* deviceExtensions[] = { VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME };﻿        deviceInfo.enabledExtensionCount = 1;﻿        deviceInfo.ppEnabledExtensionNames = deviceExtensions;﻿        if (vkCreateDevice(physicalDevice, &deviceInfo, nullptr, &device) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create device");﻿        }﻿        vkGetDeviceQueue(device, 0, 0, &graphicsQueue);﻿        vkGetDeviceQueue(device, 0, 0, &presentQueue);﻿        // Create Android surface﻿        VkAndroidSurfaceCreateInfoKHR surfaceInfo{};﻿        surfaceInfo.sType = VK\_STRUCTURE\_TYPE\_ANDROID\_SURFACE\_CREATE\_INFO\_KHR;﻿        surfaceInfo.window = (ANativeWindow\*)nwh;﻿        VkSurfaceKHR surface;﻿        if (vkCreateAndroidSurfaceKHR(instance, &surfaceInfo, nullptr, &surface) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create Android surface");﻿        }﻿        // Create swapchain﻿        VkSwapchainCreateInfoKHR swapchainInfo{};﻿        s[wapchainInfo.sTyp](wapchainInfo.sTyp#wapchainInfo.sTyp)e = VK\_STRUCTURE\_TYPE\_SWAPCHAIN\_CREATE\_INFO\_KHR;﻿        s[wapchainInfo.surf](wapchainInfo.surf#wapchainInfo.surf)ace = surface;﻿        s[wapchainInfo.minI](wapchainInfo.minI#wapchainInfo.minI)mageCount = 3;﻿        s[wapchainInfo.imag](wapchainInfo.imag#wapchainInfo.imag)eFormat = VK\_FORMAT\_B8G8R8A8\_UNORM;﻿        s[wapchainInfo.imag](wapchainInfo.imag#wapchainInfo.imag)eColorSpace = VK\_COLOR\_SPACE\_SRGB\_NONLINEAR\_KHR;﻿        s[wapchainInfo.imag](wapchainInfo.imag#wapchainInfo.imag)eExtent = { width, height };﻿        s[wapchainInfo.imag](wapchainInfo.imag#wapchainInfo.imag)eArrayLayers = 1;﻿        s[wapchainInfo.imag](wapchainInfo.imag#wapchainInfo.imag)eUsage = VK\_IMAGE\_USAGE\_COLOR\_ATTACHMENT\_BIT;﻿        s[wapchainInfo.preT](wapchainInfo.preT#wapchainInfo.preT)ransform = VK\_SURFACE\_TRANSFORM\_IDENTITY\_BIT\_KHR;﻿[swapchainInfo.com](swapchainInfo.com#swapchainInfo.com)positeAlpha = VK\_COMPOSITE\_ALPHA\_OPAQUE\_BIT\_KHR;﻿        s[wapchainInfo.pres](wapchainInfo.pres#wapchainInfo.pres)entMode = VK\_PRESENT\_MODE\_FIFO\_KHR;﻿        if (vkCreateSwapchainKHR(device, &swapchainInfo, nullptr, &swapchain) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create swapchain");﻿        }﻿        // Get swapchain images﻿        uint32\_t imageCount = 0;﻿        vkGetSwapchainImagesKHR(device, swapchain, &imageCount, nullptr);﻿        s[wapchainImages.resi](wapchainImages.resi#wapchainImages.resi)ze(imageCount);﻿        s[wapchainImageViews.resi](wapchainImageViews.resi#wapchainImageViews.resi)ze(imageCount);﻿        vkGetSwapchainImagesKHR(device, swapchain, &imageCount, s[wapchainImages.data](wapchainImages.data#wapchainImages.data)());﻿        for (uint32\_t i = 0; i < imageCount; i++) {﻿            VkImageViewCreateInfo viewInfo{};﻿            viewInfo.sType = VK\_STRUCTURE\_TYPE\_IMAGE\_VIEW\_CREATE\_INFO;﻿            viewInfo.image = swapchainImages[i];﻿            viewInfo.viewType = VK\_IMAGE\_VIEW\_TYPE\_2D;﻿            viewInfo.format = VK\_FORMAT\_B8G8R8A8\_UNORM;﻿            viewInfo.subresourceRange.aspectMask = VK\_IMAGE\_ASPECT\_COLOR\_BIT;﻿            viewInfo.subresourceRange.levelCount = 1;﻿            viewInfo.subresourceRange.layerCount = 1;﻿            if (vkCreateImageView(device, &viewInfo, nullptr, &swapchainImageViews[i]) != VK\_SUCCESS) {﻿                throw std::runtime\_error("Failed to create image view");﻿            }﻿        }﻿        // Create render pass﻿        VkAttachmentDescription colorAttachment{};﻿        colorAttachment.format = VK\_FORMAT\_B8G8R8A8\_UNORM;﻿        colorAttachment.samples = VK\_SAMPLE\_COUNT\_1\_BIT;﻿        colorAttachment.loadOp = VK\_ATTACHMENT\_LOAD\_OP\_CLEAR;﻿        colorAttachment.storeOp = VK\_ATTACHMENT\_STORE\_OP\_STORE;﻿        colorAttachment.stencilLoadOp = VK\_ATTACHMENT\_LOAD\_OP\_DONT\_CARE;﻿        colorAttachment.stencilStoreOp = VK\_ATTACHMENT\_STORE\_OP\_DONT\_CARE;﻿        colorAttachment.initialLayout = VK\_IMAGE\_LAYOUT\_UNDEFINED;﻿        colorAttachment.finalLayout = VK\_IMAGE\_LAYOUT\_PRESENT\_SRC\_KHR;﻿        VkAttachmentReference colorRef{};﻿        colorRef.attachment = 0;﻿        colorRef.layout = VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL;﻿        VkSubpassDescription subpass{};﻿        subpass.pipelineBindPoint = VK\_PIPELINE\_BIND\_POINT\_GRAPHICS;﻿        subpass.colorAttachmentCount = 1;﻿        subpass.pColorAttachments = &colorRef;﻿        VkRenderPassCreateInfo renderPassInfo{};﻿        renderPassInfo.sType = VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_CREATE\_INFO;﻿        renderPassInfo.attachmentCount = 1;﻿        renderPassInfo.pAttachments = &colorAttachment;﻿        renderPassInfo.subpassCount = 1;﻿        renderPassInfo.pSubpasses = &subpass;﻿        if (vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create render pass");﻿        }﻿        // Create descriptor set layout﻿        VkDescriptorSetLayoutBinding samplerLayoutBinding{};﻿        samplerLayoutBinding.binding = 0;﻿        samplerLayoutBinding.descriptorType = VK\_DESCRIPTOR\_TYPE\_COMBINED\_IMAGE\_SAMPLER;﻿        samplerLayoutBinding.descriptorCount = 1;﻿        samplerLayoutBinding.stageFlags = VK\_SHADER\_STAGE\_FRAGMENT\_BIT;﻿        VkDescriptorSetLayoutCreateInfo layoutInfo{};﻿        layoutInfo.sType = VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_SET\_LAYOUT\_CREATE\_INFO;﻿        layoutInfo.bindingCount = 1;﻿        layoutInfo.pBindings = &samplerLayoutBinding;﻿        if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create descriptor set layout");﻿        }﻿        // Create pipeline﻿        VkShaderModule vertShader, fragShader;﻿        // Placeholder: Load vs\_chunk.spv, fs\_chunk.spv (compiled shaders)﻿        VkPipelineShaderStageCreateInfo shaderStages[2]{};﻿        shaderStages[0].sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_SHADER\_STAGE\_CREATE\_INFO;﻿        shaderStages[0].stage = VK\_SHADER\_STAGE\_VERTEX\_BIT;﻿        shaderStages[0].module = vertShader;﻿        shaderStages[0].pName = "main";﻿        shaderStages[1].sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_SHADER\_STAGE\_CREATE\_INFO;﻿        shaderStages[1].stage = VK\_SHADER\_STAGE\_FRAGMENT\_BIT;﻿        shaderStages[1].module = fragShader;﻿        shaderStages[1].pName = "main";﻿        VkVertexInputBindingDescription bindingDesc{};﻿        bindingDesc.binding = 0;﻿        bindingDesc.stride = 6 \* sizeof(float); // pos (3), uv (2), instance idx (1)﻿        bindingDesc.inputRate = VK\_VERTEX\_INPUT\_RATE\_VERTEX;﻿        VkVertexInputAttributeDescription attrDesc[3]{};﻿        attrDesc[0].binding = 0;﻿        attrDesc[0].location = 0;﻿        attrDesc[0].format = VK\_FORMAT\_R32G32B32\_SFLOAT;﻿        attrDesc[0].offset = 0;﻿        attrDesc[1].binding = 0;﻿        attrDesc[1].location = 1;﻿        attrDesc[1].format = VK\_FORMAT\_R32G32\_SFLOAT;﻿        attrDesc[1].offset = 3 \* sizeof(float);﻿        attrDesc[2].binding = 0;﻿        attrDesc[2].location = 2;﻿        attrDesc[2].format = VK\_FORMAT\_R32\_SFLOAT;﻿        attrDesc[2].offset = 5 \* sizeof(float);﻿        VkPipelineVertexInputStateCreateInfo vertexInputInfo{};﻿        vertexInputInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_VERTEX\_INPUT\_STATE\_CREATE\_INFO;﻿        vertexInputInfo.vertexBindingDescriptionCount = 1;﻿        vertexInputInfo.pVertexBindingDescriptions = &bindingDesc;﻿        vertexInputInfo.vertexAttributeDescriptionCount = 3;﻿        vertexInputInfo.pVertexAttributeDescriptions = attrDesc;﻿        VkPipelineInputAssemblyStateCreateInfo inputAssembly{};﻿        inputAssembly.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_INPUT\_ASSEMBLY\_STATE\_CREATE\_INFO;﻿        inputAssembly.topology = VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_LIST;﻿        inputAssembly.primitiveRestartEnable = VK\_FALSE;﻿        VkViewport viewport{};﻿        viewport.x = 0.0f;﻿        viewport.y = 0.0f;﻿        viewport.width = (float)width;﻿        viewport.height = (float)height;﻿        viewport.minDepth = 0.0f;﻿        viewport.maxDepth = 1.0f;﻿        VkRect2D scissor{};﻿        scissor.offset = {0, 0};﻿        scissor.extent = {width, height};﻿        VkPipelineViewportStateCreateInfo viewportState{};﻿        viewportState.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_VIEWPORT\_STATE\_CREATE\_INFO;﻿        viewportState.viewportCount = 1;﻿        viewportState.pViewports = &viewport;﻿        viewportState.scissorCount = 1;﻿        viewportState.pScissors = &scissor;﻿        VkPipelineRasterizationStateCreateInfo rasterizer{};﻿        rasterizer.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_RASTERIZATION\_STATE\_CREATE\_INFO;﻿        rasterizer.polygonMode = VK\_POLYGON\_MODE\_FILL;﻿        rasterizer.lineWidth = 1.0f;﻿        rasterizer.cullMode = VK\_CULL\_MODE\_BACK\_BIT;﻿[rasterizer.fr](rasterizer.fr#rasterizer.fr)ontFace = VK\_FRONT\_FACE\_COUNTER\_CLOCKWISE;﻿        VkPipelineMultisampleStateCreateInfo multisampling{};﻿        multisampling.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_MULTISAMPLE\_STATE\_CREATE\_INFO;﻿        multisampling.rasterizationSamples = VK\_SAMPLE\_COUNT\_1\_BIT;﻿        VkPipelineColorBlendAttachmentState colorBlendAttachment{};﻿        colorBlendAttachment.colorWriteMask = VK\_COLOR\_COMPONENT\_R\_BIT | VK\_COLOR\_COMPONENT\_G\_BIT |﻿                                              VK\_COLOR\_COMPONENT\_B\_BIT | VK\_COLOR\_COMPONENT\_A\_BIT;﻿        colorBlendAttachment.blendEnable = VK\_FALSE;﻿        VkPipelineColorBlendStateCreateInfo colorBlending{};﻿        colorBlending.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_COLOR\_BLEND\_STATE\_CREATE\_INFO;﻿        colorBlending.attachmentCount = 1;﻿        colorBlending.pAttachments = &colorBlendAttachment;﻿        VkPipelineLayoutCreateInfo pipelineLayoutInfo{};﻿        pipelineLayoutInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_LAYOUT\_CREATE\_INFO;﻿        pipelineLayoutInfo.setLayoutCount = 1;﻿        pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;﻿        if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create pipeline layout");﻿        }﻿        VkGraphicsPipelineCreateInfo pipelineInfo{};﻿        pipelineInfo.sType = VK\_STRUCTURE\_TYPE\_GRAPHICS\_PIPELINE\_CREATE\_INFO;﻿        pipelineInfo.stageCount = 2;﻿        pipelineInfo.pStages = shaderStages;﻿        pipelineInfo.pVertexInputState = &vertexInputInfo;﻿        pipelineInfo.pInputAssemblyState = &inputAssembly;﻿        pipelineInfo.pViewportState = &viewportState;﻿        pipelineInfo.pRasterizationState = &rasterizer;﻿        pipelineInfo.pMultisampleState = &multisampling;﻿        pipelineInfo.pColorBlendState = &colorBlending;﻿        pipelineInfo.layout = pipelineLayout;﻿        pipelineInfo.renderPass = renderPass;﻿        pipelineInfo.subpass = 0;﻿        if (vkCreateGraphicsPipelines(device, VK\_NULL\_HANDLE, 1, &pipelineInfo, nullptr, &pipeline) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create pipeline");﻿        }﻿        // Create framebuffers﻿        framebuffers.resize(s[wapchainImageViews.size](wapchainImageViews.size#wapchainImageViews.size)());﻿        for (size\_t i = 0; i < s[wapchainImageViews.size](wapchainImageViews.size#wapchainImageViews.size)(); i++) {﻿            VkFramebufferCreateInfo fbInfo{};﻿            fbInfo.sType = VK\_STRUCTURE\_TYPE\_FRAMEBUFFER\_CREATE\_INFO;﻿            fbInfo.renderPass = renderPass;﻿            fbInfo.attachmentCount = 1;﻿            fbInfo.pAttachments = &swapchainImageViews[i];﻿            fbInfo.width = width;﻿            fbInfo.height = height;﻿            fbInfo.layers = 1;﻿            if (vkCreateFramebuffer(device, &fbInfo, nullptr, &framebuffers[i]) != VK\_SUCCESS) {﻿                throw std::runtime\_error("Failed to create framebuffer");﻿            }﻿        }﻿        // Create command pool﻿        VkCommandPoolCreateInfo poolInfo{};﻿        poolInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_POOL\_CREATE\_INFO;﻿        poolInfo.queueFamilyIndex = 0; // Simplified﻿        if (vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to create command pool");﻿        }﻿        // Allocate command buffers﻿        commandBuffers.resize(framebuffers.size());﻿        VkCommandBufferAllocateInfo allocInfo{};﻿        allocInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_BUFFER\_ALLOCATE\_INFO;﻿[allocInfo.com](allocInfo.com#allocInfo.com)mandPool = commandPool;﻿        allocInfo.level = VK\_COMMAND\_BUFFER\_LEVEL\_PRIMARY;﻿[allocInfo.com](allocInfo.com#allocInfo.com)mandBufferCount = (uint32\_t)commandBuffers.size();﻿        if (vkAllocateCommandBuffers(device, &allocInfo, commandBuffers.data()) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to allocate command buffers");﻿        }﻿        // Create texture (placeholder)﻿        // Load atlas.png into textureImage, textureImageMemory, textureImageView, textureSampler﻿    } catch (const std::exception& e) {﻿        printf("Renderer init failed: %s\n", e.what());﻿    }﻿}﻿void renderer\_shutdown() noexcept {﻿    for (auto fb : framebuffers) vkDestroyFramebuffer(device, fb, nullptr);﻿    vkDestroyCommandPool(device, commandPool, nullptr);﻿    vkDestroyPipeline(device, pipeline, nullptr);﻿    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);﻿    vkDestroyRenderPass(device, renderPass, nullptr);﻿    vkDestroySwapchainKHR(device, swapchain, nullptr);﻿    for (auto view : swapchainImageViews) vkDestroyImageView(device, view, nullptr);﻿    vkDestroyDescriptorPool(device, descriptorPool, nullptr);﻿    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);﻿    vkDestroySampler(device, textureSampler, nullptr);﻿    vkDestroyImageView(device, textureImageView, nullptr);﻿    vkDestroyImage(device, textureImage, nullptr);﻿    vkFreeMemory(device, textureImageMemory, nullptr);﻿    vkDestroyDevice(device, nullptr);﻿    vkDestroyInstance(instance, nullptr);﻿}﻿void set\_shared\_buffer(uint8\_t\* ptr, size\_t maxEntities) noexcept {﻿    sharedBuffer = ptr;﻿    jarock::maxEntities = maxEntities;﻿}﻿uint64\_t create\_mesh(const float\* verts, uint32\_t vcount, const uint16\_t\* idx, uint32\_t icount) noexcept {﻿    try {﻿        Mesh mesh{};﻿        mesh.vertexCount = vcount;﻿        mesh.indexCount = icount;﻿        createBuffer(vcount \* 6 \* sizeof(float), VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT,﻿                     VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT | VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT,﻿                     mesh.vertexBuffer, mesh.vertexMemory);﻿        void\* data;﻿        vkMapMemory(device, mesh.vertexMemory, 0, vcount \* 6 \* sizeof(float), 0, &data);﻿        memcpy(data, verts, vcount \* 6 \* sizeof(float));﻿        vkUnmapMemory(device, mesh.vertexMemory);﻿        createBuffer(icount \* sizeof(uint16\_t), VK\_BUFFER\_USAGE\_INDEX\_BUFFER\_BIT,﻿                     VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT | VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT,﻿                     mesh.indexBuffer, mesh.indexMemory);﻿        vkMapMemory(device, mesh.indexMemory, 0, icount \* sizeof(uint16\_t), 0, &data);﻿        memcpy(data, idx, icount \* sizeof(uint16\_t));﻿        vkUnmapMemory(device, mesh.indexMemory);﻿        return reinterpret\_cast<uint64\_t>(new Mesh(mesh));﻿    } catch (const std::exception& e) {﻿        printf("Create mesh failed: %s\n", e.what());﻿        return 0;﻿    }﻿}﻿void submit\_mesh(uint64\_t pack, const float\* instances, uint32\_t instanceCount, const float\* viewProj) noexcept {﻿    try {﻿        Mesh\* mesh = reinterpret\_cast<Mesh\*>(pack);﻿        uint32\_t imageIdx;﻿        vkAcquireNextImageKHR(device, swapchain, UINT64\_MAX, VK\_NULL\_HANDLE, VK\_NULL\_HANDLE, &imageIdx);﻿        VkCommandBufferBeginInfo beginInfo{};﻿        beginInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_BUFFER\_BEGIN\_INFO;﻿        beginInfo.flags = VK\_COMMAND\_BUFFER\_USAGE\_ONE\_TIME\_SUBMIT\_BIT;﻿        vkBeginCommandBuffer(commandBuffers[imageIdx], &beginInfo);﻿        VkRenderPassBeginInfo renderPassInfo{};﻿        renderPassInfo.sType = VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_BEGIN\_INFO;﻿        renderPassInfo.renderPass = renderPass;﻿[renderPassInfo.fr](renderPassInfo.fr#renderPassInfo.fr)amebuffer = framebuffers[imageIdx];﻿        renderPassInfo.renderArea.offset = {0, 0};﻿        renderPassInfo.renderArea.extent = {1280, 720};﻿        VkClearValue clearColor = {{{0.0f, 0.0f, 0.0f, 1.0f}}};﻿        renderPassInfo.clearValueCount = 1;﻿        renderPassInfo.pClearValues = &clearColor;﻿        vkCmdBeginRenderPass(commandBuffers[imageIdx], &renderPassInfo, VK\_SUBPASS\_CONTENTS\_INLINE);﻿        vkCmdBindPipeline(commandBuffers[imageIdx], VK\_PIPELINE\_BIND\_POINT\_GRAPHICS, pipeline);﻿        VkBuffer vertexBuffers[] = { mesh->vertexBuffer };﻿        VkDeviceSize offsets[] = { 0 };﻿        vkCmdBindVertexBuffers(commandBuffers[imageIdx], 0, 1, vertexBuffers, offsets);﻿        vkCmdBindIndexBuffer(commandBuffers[imageIdx], mesh->indexBuffer, 0, VK\_INDEX\_TYPE\_UINT16);﻿        vkCmdBindDescriptorSets(commandBuffers[imageIdx], VK\_PIPELINE\_BIND\_POINT\_GRAPHICS, pipelineLayout, 0, 1, &descriptorSets[imageIdx], 0, nullptr);﻿        // Render chunks﻿        vkCmdDrawIndexed(commandBuffers[imageIdx], mesh->indexCount, instanceCount, 0, 0, 0);﻿        // Render entities from shared buffer﻿        std::vector<float> entityInstances;﻿        for (size\_t i = 0; i < maxEntities; i++) {﻿            uint32\_t id = \*(uint32\_t\*)(sharedBuffer + i \* 16);﻿            float x = \*(float\*)(sharedBuffer + i \* 16 + 4);﻿            float y = \*(float\*)(sharedBuffer + i \* 16 + 8);﻿            float z = \*(float\*)(sharedBuffer + i \* 16 + 12);﻿            entityInstances.push\_back(x); entityInstances.push\_back(y); entityInstances.push\_back(z);﻿            entityInstances.push\_back(1); entityInstances.push\_back(1); entityInstances.push\_back(1);﻿        }﻿        if (!entityInstances.empty()) {﻿            VkBuffer instanceBuffer;﻿            VkDeviceMemory instanceMemory;﻿            createBuffer(entityInstances.size() \* sizeof(float), VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT,﻿                         VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT | VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT,﻿                         instanceBuffer, instanceMemory);﻿            void\* data;﻿            vkMapMemory(device, instanceMemory, 0, entityInstances.size() \* sizeof(float), 0, &data);﻿            memcpy(data, entityInstances.data(), entityInstances.size() \* sizeof(float));﻿            vkUnmapMemory(device, instanceMemory);﻿            VkBuffer instanceBuffers[] = { instanceBuffer };﻿            vkCmdBindVertexBuffers(commandBuffers[imageIdx], 1, 1, instanceBuffers, offsets);﻿            vkCmdDrawIndexed(commandBuffers[imageIdx], mesh->indexCount, entityInstances.size() / 6, 0, 0, 0);﻿            vkDestroyBuffer(device, instanceBuffer, nullptr);﻿            vkFreeMemory(device, instanceMemory, nullptr);﻿        }﻿        vkCmdEndRenderPass(commandBuffers[imageIdx]);﻿        if (vkEndCommandBuffer(commandBuffers[imageIdx]) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to record command buffer");﻿        }﻿        VkSubmitInfo submitInfo{};﻿        submitInfo.sType = VK\_STRUCTURE\_TYPE\_SUBMIT\_INFO;﻿[submitInfo.com](submitInfo.com#submitInfo.com)mandBufferCount = 1;﻿        submitInfo.pCommandBuffers = &commandBuffers[imageIdx];﻿        if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK\_NULL\_HANDLE) != VK\_SUCCESS) {﻿            throw std::runtime\_error("Failed to submit draw command");﻿        }﻿        VkPresentInfoKHR presentInfo{};﻿        presentInfo.sType = VK\_STRUCTURE\_TYPE\_PRESENT\_INFO\_KHR;﻿        presentInfo.swapchainCount = 1;﻿        presentInfo.pSwapchains = &swapchain;﻿        presentInfo.pImageIndices = &imageIdx;﻿        vkQueuePresentKHR(presentQueue, &presentInfo);﻿    } catch (const std::exception& e) {﻿        printf("Render failed: %s\n", e.what());﻿    }﻿}﻿}﻿#version 450﻿layout(location = 0) in vec3 inPos;﻿layout(location = 1) in vec2 inUV;﻿layout(location = 2) in float inInstanceIdx;﻿layout(location = 0) out vec2 fragUV;﻿layout(binding = 0) uniform UBO { mat4 viewProj; } ubo;﻿void main() {﻿    gl\_Position = ubo.viewProj \* vec4(inPos + vec3(0, 0, inInstanceIdx), 1.0);﻿    fragUV = inUV;﻿}﻿#version 450﻿layout(location = 0) in vec2 fragUV;﻿layout(location = 0) out vec4 outColor;﻿layout(binding = 0) uniform sampler2D texSampler;﻿void main() {﻿    outColor = texture(texSampler, fragUV);﻿}