**Minecarft Jarock edition**

public class JarockGame {﻿    static {﻿        // Load Rust/C++ library﻿        System.loadLibrary("jarock\_performance");﻿    }﻿    public native void updateEntities(); // gọi Rust/C++ để xử lý AI & physics﻿    public native void renderFrame();    // gọi Vulkan render﻿    public void gameLoop() {﻿        while (true) {﻿            updateEntities();  // update logic﻿            renderFrame();     // render graphics﻿            try { Thread.sleep(16); } catch (InterruptedException e) {} // ~60 FPS﻿        }﻿    }﻿    public static void main(String[] args) {﻿        new JarockGame().gameLoop();﻿    }﻿}﻿#[no\_mangle]﻿pub extern "C" fn update\_entities() {﻿    // xử lý AI mobs, spawn logic, physics﻿    println!("Updating entities in Rust");﻿}﻿#[no\_mangle]﻿pub extern "C" fn render\_frame() {﻿    // gọi Vulkan render pipeline﻿    println!("Rendering frame with Vulkan in Rust");﻿}﻿#include <vulkan/vulkan.h>﻿#include <iostream>﻿extern "C" void init\_vulkan() {﻿    std::cout << "Vulkan initialized" << std::endl;﻿    // setup Vulkan instance, device, swapchain (chi tiết dài )﻿}﻿using System;﻿class MapEditor {﻿    static void Main() {﻿        Console.WriteLine("Jarock Map Editor started");﻿        // GUI + editor logic﻿    }﻿}﻿function spawnMob(type, x, y, z) {﻿    console.log(`Spawning mob ${type} at ${x},${y},${z}`);﻿    // sẽ gọi qua Java core game loop API﻿}﻿module.exports = { spawnMob };﻿use std::collections::HashMap;﻿#[derive(Clone)]﻿pub struct Mob {﻿    pub id: u32,﻿    pub x: f32,﻿    pub y: f32,﻿    pub z: f32,﻿    pub vx: f32,﻿    pub vy: f32,﻿    pub vz: f32,﻿}﻿static mut MOBS: Option<HashMap<u32, Mob>> = None;﻿#[no\_mangle]﻿pub extern "C" fn init\_entities() {﻿    unsafe { MOBS = Some(HashMap::new()); }﻿}﻿#[no\_mangle]﻿pub extern "C" fn spawn\_mob(id: u32, x: f32, y: f32, z: f32) {﻿    let mob = Mob { id, x, y, z, vx:0.0, vy:0.0, vz:0.0 };﻿    unsafe { MOBS.as\_mut().unwrap().insert(id, mob); }﻿}﻿#[no\_mangle]﻿pub extern "C" fn update\_entities() {﻿    unsafe {﻿        if let Some(mobs) = &mut MOBS {﻿            for mob in mobs.values\_mut() {﻿                // Vật lý đơn giản﻿                mob.vy -= 0.01; // gravity﻿                mob.x += mob.vx;﻿                mob.y += mob.vy;﻿                mob.z += mob.vz;﻿                if mob.y < 0.0 { mob.y = 0.0; mob.vy = 0.0; } // ground collision﻿            }﻿        }﻿    }﻿}﻿#[no\_mangle]﻿pub extern "C" fn render\_frame() {﻿    unsafe {﻿        // gọi C++ Vulkan render﻿        extern "C" { fn render\_vulkan\_frame(); }﻿        render\_vulkan\_frame();﻿    }﻿}﻿#include <vulkan/vulkan.h>﻿#include <iostream>﻿#include <vector>﻿VkInstance instance;﻿VkDevice device;﻿VkQueue graphicsQueue;﻿VkSurfaceKHR surface;﻿extern "C" void init\_vulkan() {﻿    std::cout << "Vulkan Init start" << std::endl;﻿    // 1. Create Vulkan Instance﻿    VkApplicationInfo appInfo{};﻿    appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;﻿    appInfo.pApplicationName = "Jarock Edition";﻿    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\_3;﻿    VkInstanceCreateInfo createInfo{};﻿    createInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;﻿    createInfo.pApplicationInfo = &appInfo;﻿    if(vkCreateInstance(&createInfo, nullptr, &instance) != VK\_SUCCESS){﻿        std::cerr << "Failed to create Vulkan instance!" << std::endl;﻿        return;﻿    }﻿    std::cout << "Vulkan Instance created!" << std::endl;﻿    // TODO: Device, Swapchain, Shader setup﻿}﻿extern "C" void render\_vulkan\_frame() {﻿    // TODO: Clear screen, draw mobs/items﻿    std::cout << "Vulkan frame render called" << std::endl;﻿}﻿public class JarockGame {﻿    static { System.loadLibrary("jarock\_performance"); }﻿    public native void initEntities();﻿    public native void spawnMob(int id, float x, float y, float z);﻿    public native void updateEntities();﻿    public native void renderFrame();﻿    public void gameLoop() {﻿        while(true) {﻿            // TODO: xử lý input, network sync﻿            updateEntities(); // AI + Physics﻿            renderFrame();    // Vulkan﻿            try { Thread.sleep(16); } catch(Exception e){}﻿        }﻿    }﻿    public static void main(String[] args) {﻿        JarockGame game = new JarockGame();﻿        game.initEntities();﻿        game.spawnMob(1, 0, 10, 0); // spawn thử mob﻿        game.gameLoop();﻿    }﻿}﻿const { spawnMob } = require("jarock\_api"); // giả sử wrapper JNI/Node-GraalVM﻿spawnMob(2, 5, 15, -3); // gọi Java game loop để spawn mob﻿#include <vulkan/vulkan.h>﻿#include <vector>﻿#include <iostream>﻿// Core Vulkan objects﻿VkInstance instance;﻿VkPhysicalDevice physicalDevice = VK\_NULL\_HANDLE;﻿VkDevice device;﻿VkQueue graphicsQueue;﻿VkSurfaceKHR surface;﻿VkSwapchainKHR swapChain;﻿VkFormat swapChainImageFormat;﻿VkExtent2D swapChainExtent;﻿std::vector<VkImage> swapChainImages;﻿VkRenderPass renderPass;﻿VkPipelineLayout pipelineLayout;﻿VkPipeline graphicsPipeline;﻿extern "C" void init\_vulkan() {﻿    // --- Vulkan Instance ---﻿    VkApplicationInfo appInfo{};﻿    appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;﻿    appInfo.pApplicationName = "Jarock Edition";﻿    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\_3;﻿    VkInstanceCreateInfo createInfo{};﻿    createInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;﻿    createInfo.pApplicationInfo = &appInfo;﻿    if(vkCreateInstance(&createInfo, nullptr, &instance) != VK\_SUCCESS){﻿        std::cerr << "Failed to create Vulkan instance!" << std::endl;﻿        return;﻿    }﻿    // --- Pick Physical Device ---﻿    uint32\_t deviceCount = 0;﻿    vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);﻿    std::vector<VkPhysicalDevice> devices(deviceCount);﻿    vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());﻿    physicalDevice = devices[0];﻿    // --- Logical Device & Graphics Queue ---﻿    float queuePriority = 1.0f;﻿    VkDeviceQueueCreateInfo queueCreateInfo{};﻿    queueCreateInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_QUEUE\_CREATE\_INFO;﻿    queueCreateInfo.queueFamilyIndex = 0;﻿    queueCreateInfo.queueCount = 1;﻿    queueCreateInfo.pQueuePriorities = &queuePriority;﻿    VkDeviceCreateInfo deviceCreateInfo{};﻿    deviceCreateInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_CREATE\_INFO;﻿    deviceCreateInfo.queueCreateInfoCount = 1;﻿    deviceCreateInfo.pQueueCreateInfos = &queueCreateInfo;﻿    if(vkCreateDevice(physicalDevice, &deviceCreateInfo, nullptr, &device) != VK\_SUCCESS){﻿        std::cerr << "Failed to create logical device!" << std::endl;﻿        return;﻿    }﻿    vkGetDeviceQueue(device, 0, 0, &graphicsQueue);﻿    std::cout << "Vulkan initialized (Instance + Device + Queue)." << std::endl;﻿    // TODO: Setup Swapchain, RenderPass, Pipeline, Framebuffers, Shaders﻿}﻿extern "C" void render\_vulkan\_frame(float\* mob\_positions, int mob\_count) {﻿    // TODO: Transfer mob\_positions to GPU buffers (vertex buffer)﻿    // Clear framebuffer, bind pipeline, draw mobs﻿﻿}﻿use std::collections::HashMap;﻿#[derive(Clone)]﻿pub struct Mob {﻿    pub id: u32,﻿    pub x: f32,﻿    pub y: f32,﻿    pub z: f32,﻿    pub vx: f32,﻿    pub vy: f32,﻿    pub vz: f32,﻿}﻿static mut MOBS: Option<HashMap<u32, Mob>> = None;﻿#[no\_mangle]﻿pub extern "C" fn init\_entities() {﻿    unsafe { MOBS = Some(HashMap::new()); }﻿}﻿#[no\_mangle]﻿pub extern "C" fn spawn\_mob(id: u32, x: f32, y: f32, z: f32) {﻿    let mob = Mob { id, x, y, z, vx:0.0, vy:0.0, vz:0.0 };﻿    unsafe { MOBS.as\_mut().unwrap().insert(id, mob); }﻿}﻿#[no\_mangle]﻿pub extern "C" fn update\_entities(player\_positions: \*const (f32,f32,f32), player\_count: usize) {﻿    unsafe {﻿        if let Some(mobs) = &mut MOBS {﻿            for mob in mobs.values\_mut() {﻿                // --- Gravity ---﻿                mob.vy -= 0.01;﻿                // --- Ground Collision ---﻿                if mob.y + mob.vy < 0.0 { mob.y = 0.0; mob.vy = 0.0; }﻿                else { mob.y += mob.vy; }﻿                // --- AI: Follow nearest player ---﻿                if player\_count > 0 {﻿                    let mut nearest = 0;﻿                    let mut min\_dist = f32::MAX;﻿                    for i in 0..player\_count {﻿                        let (px, py, pz) = \*player\_positions.add(i);﻿                        let dist = ((mob.x-px).powi(2) + (mob.y-py).powi(2) + (mob.z-pz).powi(2)).sqrt();﻿                        if dist < min\_dist { min\_dist = dist; nearest = i; }﻿                    }﻿                    let (tx, ty, tz) = \*player\_positions.add(nearest);﻿                    let dx = tx - mob.x;﻿                    let dz = tz - mob.z;﻿                    let len = (dx\*dx + dz\*dz).sqrt();﻿                    if len > 0.0 {﻿                        mob.vx = dx / len \* 0.05;﻿                        mob.vz = dz / len \* 0.05;﻿                    }﻿                    mob.x += mob.vx;﻿                    mob.z += mob.vz;﻿                }﻿            }﻿        }﻿    }﻿}﻿#[no\_mangle]﻿pub extern "C" fn render\_frame() {﻿    extern "C" { fn render\_vulkan\_frame(mob\_positions: \*mut f32, mob\_count: i32); }﻿    unsafe {﻿        if let Some(mobs) = &MOBS {﻿            let mut buffer: Vec<f32> = vec![];﻿            for mob in mobs.values() {﻿                buffer.push(mob.x); buffer.push(mob.y); buffer.push(mob.z);﻿            }﻿            render\_vulkan\_frame(buffer.as\_mut\_ptr(), mobs.len() as i32);﻿        }﻿    }﻿}﻿import java.util.\*;﻿public class JarockGame {﻿    static { System.loadLibrary("jarock\_performance"); }﻿    public native void initEntities();﻿    public native void spawnMob(int id, float x, float y, float z);﻿    public native void updateEntities(float[] playerPositions, int playerCount);﻿    public native void renderFrame();﻿    List<float[]> players = new ArrayList<>();﻿    public void handleInput() {﻿        // TODO: đọc keyboard/mouse﻿    }﻿    public float[] flattenPlayerPositions() {﻿        float[] arr = new float[players.size()\*3];﻿        for(int i=0;i<players.size();i++){﻿            float[] p = players.get(i);﻿            arr[i\*3]=p[0]; arr[i\*3+1]=p[1]; arr[i\*3+2]=p[2];﻿        }﻿        return arr;﻿    }﻿    public void gameLoop() {﻿        while(true) {﻿            handleInput(); ﻿            updateEntities(flattenPlayerPositions(), players.size());﻿            renderFrame();﻿            try { Thread.sleep(16); } catch(Exception e){}﻿        }﻿    }﻿    public static void main(String[] args) {﻿        JarockGame game = new JarockGame();﻿        game.initEntities();﻿        game.spawnMob(1, 0, 10, 0);﻿        game.players.add(new float[]{5f,0f,5f}); // demo player﻿        game.gameLoop();﻿    }﻿}﻿const jarock = require("jarock\_api"); // giả sử wrapper JNI﻿// Gọi spawn mob vào Java -> Rust game loop﻿jarock.spawnMob(2, 10, 5, -3);﻿#include <vulkan/vulkan.h>﻿#include <vector>﻿#include <iostream>﻿VkInstance instance;﻿VkPhysicalDevice physicalDevice = VK\_NULL\_HANDLE;﻿VkDevice device;﻿VkQueue graphicsQueue;﻿VkSwapchainKHR swapChain;﻿VkRenderPass renderPass;﻿VkPipelineLayout pipelineLayout;﻿VkPipeline graphicsPipeline;﻿extern "C" void init\_vulkan() {﻿    // --- Vulkan Instance ---﻿    VkApplicationInfo appInfo{};﻿    appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;﻿    appInfo.pApplicationName = "Jarock Edition";﻿    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\_3;﻿    VkInstanceCreateInfo createInfo{};﻿    createInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;﻿    createInfo.pApplicationInfo = &appInfo;﻿    if(vkCreateInstance(&createInfo, nullptr, &instance) != VK\_SUCCESS){﻿        std::cerr << "Failed to create Vulkan instance!\n"; return;﻿    }﻿    // --- Pick Physical Device ---﻿    uint32\_t deviceCount = 0;﻿    vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);﻿    std::vector<VkPhysicalDevice> devices(deviceCount);﻿    vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());﻿    physicalDevice = devices[0];﻿    // --- Logical Device & Queue ---﻿    float queuePriority = 1.0f;﻿    VkDeviceQueueCreateInfo queueInfo{};﻿    queueInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_QUEUE\_CREATE\_INFO;﻿    queueInfo.queueFamilyIndex = 0;﻿    queueInfo.queueCount = 1;﻿    queueInfo.pQueuePriorities = &queuePriority;﻿    VkDeviceCreateInfo deviceCreateInfo{};﻿    deviceCreateInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_CREATE\_INFO;﻿    deviceCreateInfo.queueCreateInfoCount = 1;﻿    deviceCreateInfo.pQueueCreateInfos = &queueInfo;﻿    if(vkCreateDevice(physicalDevice, &deviceCreateInfo, nullptr, &device)!=VK\_SUCCESS){﻿        std::cerr<<"Failed to create logical device!\n"; return;﻿    }﻿    vkGetDeviceQueue(device, 0, 0, &graphicsQueue);﻿    // --- RenderPass Skeleton ---﻿    VkAttachmentDescription colorAttachment{};﻿    colorAttachment.format = VK\_FORMAT\_B8G8R8A8\_SRGB;﻿    colorAttachment.samples = VK\_SAMPLE\_COUNT\_1\_BIT;﻿    colorAttachment.loadOp = VK\_ATTACHMENT\_LOAD\_OP\_CLEAR;﻿    colorAttachment.storeOp = VK\_ATTACHMENT\_STORE\_OP\_STORE;﻿    colorAttachment.initialLayout = VK\_IMAGE\_LAYOUT\_UNDEFINED;﻿    colorAttachment.finalLayout = VK\_IMAGE\_LAYOUT\_PRESENT\_SRC\_KHR;﻿    VkAttachmentReference colorAttachmentRef{};﻿    colorAttachmentRef.attachment = 0;﻿    colorAttachmentRef.layout = VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL;﻿    VkSubpassDescription subpass{};﻿    subpass.pipelineBindPoint = VK\_PIPELINE\_BIND\_POINT\_GRAPHICS;﻿    subpass.colorAttachmentCount = 1;﻿    subpass.pColorAttachments = &colorAttachmentRef;﻿    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){﻿        std::cerr<<"Failed to create render pass!\n"; return;﻿    }﻿    std::cout<<"Vulkan RenderPass initialized.\n";﻿    // TODO: Swapchain, Framebuffers, Shaders, Pipeline﻿}﻿extern "C" void render\_vulkan\_frame(float\* mob\_positions, int mob\_count) {﻿    // TODO: Upload mob\_positions to vertex buffer﻿    // Clear framebuffer, bind pipeline, draw each mob﻿﻿}﻿#[derive(Clone)]﻿pub struct Mob {﻿    pub id: u32,﻿    pub x: f32, pub y: f32, pub z: f32,﻿    pub vx: f32, pub vy: f32, pub vz: f32,﻿}﻿static mut MOBS: Option<Vec<Mob>> = None;﻿#[no\_mangle]﻿pub extern "C" fn update\_entities(player\_positions: \*const (f32,f32,f32), player\_count: usize) {﻿    unsafe {﻿        if let Some(mobs) = &mut MOBS {﻿            for mob in mobs.iter\_mut() {﻿                // Gravity﻿                mob.vy -= 0.01;﻿                mob.y += mob.vy;﻿                if mob.y < 0.0 { mob.y = 0.0; mob.vy = 0.0; }﻿                // Collision with other mobs﻿                for other in mobs.iter() {﻿                    if mob.id != other.id {﻿                        let dx = mob.x - other.x;﻿                        let dz = mob.z - other.z;﻿                        let dist2 = dx\*dx + dz\*dz;﻿                        if dist2 < 1.0 { mob.vx += dx\*0.01; mob.vz += dz\*0.01; }﻿                    }﻿                }﻿                // AI follow nearest player﻿                if player\_count>0 {﻿                    let mut nearest=0; let mut min\_dist=f32::MAX;﻿                    for i in 0..player\_count {﻿                        let (px,py,pz) = \*player\_positions.add(i);﻿                        let dist2 = (mob.x-px).powi(2)+(mob.y-py).powi(2)+(mob.z-pz).powi(2);﻿                        if dist2 < min\_dist { min\_dist=dist2; nearest=i; }﻿                    }﻿                    let (tx, ty, tz) = \*player\_positions.add(nearest);﻿                    let dx = tx - mob.x; let dz = tz - mob.z;﻿                    let len = (dx\*dx + dz\*dz).sqrt();﻿                    if len>0.0 { mob.vx = dx/len\*0.05; mob.vz = dz/len\*0.05; }﻿                    mob.x += mob.vx; mob.z += mob.vz;﻿                }﻿            }﻿        }﻿    }﻿}﻿import java.awt.event.\*;﻿import javax.swing.\*;﻿import java.util.\*;﻿public class JarockGame extends JFrame implements KeyListener, MouseListener {﻿    static { System.loadLibrary("jarock\_performance"); }﻿    public native void initEntities();﻿    public native void spawnMob(int id, float x, float y, float z);﻿    public native void updateEntities(float[] playerPositions, int playerCount);﻿    public native void renderFrame();﻿    List<float[]> players = new ArrayList<>();﻿    Set<Integer> keysPressed = new HashSet<>();﻿    public JarockGame() {﻿        addKeyListener(this); addMouseListener(this);﻿    }﻿    public void handleInput() {﻿        // simple WASD movement demo﻿        for(float[] p: players){﻿            if(keysPressed.contains(KeyEvent.VK\_W)) p[2]+=0.1f;﻿            if(keysPressed.contains(KeyEvent.VK\_S)) p[2]-=0.1f;﻿            if(keysPressed.contains(KeyEvent.VK\_A)) p[0]-=0.1f;﻿            if(keysPressed.contains(KeyEvent.VK\_D)) p[0]+=0.1f;﻿        }﻿    }﻿    public float[] flattenPlayerPositions() {﻿        float[] arr = new float[players.size()\*3];﻿        for(int i=0;i<players.size();i++){﻿            float[] p=players.get(i);﻿            arr[i\*3]=p[0]; arr[i\*3+1]=p[1]; arr[i\*3+2]=p[2];﻿        }﻿        return arr;﻿    }﻿    public void gameLoop() {﻿        while(true){﻿            handleInput();﻿            updateEntities(flattenPlayerPositions(), players.size());﻿            renderFrame();﻿            // TODO: Network sync with multiplayer clients﻿            try{ Thread.sleep(16); } catch(Exception e){}﻿        }﻿    }﻿    // KeyListener methods﻿    public void keyPressed(KeyEvent e){ keysPressed.add(e.getKeyCode()); }﻿    public void keyReleased(KeyEvent e){ keysPressed.remove(e.getKeyCode()); }﻿    public void keyTyped(KeyEvent e){}﻿    public void mousePressed(MouseEvent e){}﻿    public void mouseReleased(MouseEvent e){}﻿    public void mouseClicked(MouseEvent e){}﻿    public void mouseEntered(MouseEvent e){}﻿    public void mouseExited(MouseEvent e){}﻿    public static void main(String[] args){﻿        JarockGame game=new JarockGame();﻿        game.initEntities();﻿        game.spawnMob(1,0,10,0);﻿        game.players.add(new float[]{0f,0f,0f});﻿        game.gameLoop();﻿    }﻿}﻿#[derive(Clone, Copy)]﻿pub struct Block {﻿    pub id: u8, // 0=air, 1=stone, 2=dirt...﻿}﻿pub struct Chunk {﻿    pub x: i32,﻿    pub y: i32,﻿    pub z: i32,﻿    pub blocks: Vec<Block>,﻿}﻿static mut CHUNKS: Vec<Chunk> = Vec::new();﻿#[no\_mangle]﻿pub extern "C" fn create\_chunk(x:i32, y:i32, z:i32) {﻿    let mut blocks = vec![Block{id:0}; 16\*16\*16];﻿    // Fill some blocks for demo﻿    for i in 0..16 { for j in 0..16 { for k in 0..16 {﻿        if j==0 { blocks[i+16\*(j+16\*k)].id = 1; } // ground layer﻿    }}}﻿    unsafe { CHUNKS.push(Chunk{x,y,z,blocks}); }﻿}﻿#[no\_mangle]﻿pub extern "C" fn update\_chunk(chunk\_index: usize) {﻿    unsafe {﻿        if let Some(chunk) = CHUNKS.get\_mut(chunk\_index) {﻿            // Example: simple physics or block updates﻿            // 10% processing: e.g., grow grass﻿            for block in chunk.blocks.iter\_mut() {﻿                if block.id == 0 { block.id = 0; } // placeholder﻿            }﻿        }﻿    }﻿}﻿#include <vulkan/vulkan.h>﻿#include <vector>﻿#include <iostream>﻿struct Vec3 { float x,y,z; };﻿struct BlockVertex { Vec3 pos; float color[3]; };﻿struct ChunkMesh {﻿    std::vector<BlockVertex> vertices;﻿    VkBuffer vertexBuffer;﻿    VkDeviceMemory vertexMemory;﻿};﻿std::vector<ChunkMesh> loadedChunks;﻿extern "C" void render\_chunk(int chunk\_index) {﻿    if(chunk\_index >= loadedChunks.size()) return;﻿    ChunkMesh &mesh = loadedChunks[chunk\_index];﻿    // TODO: upload mesh.vertices to GPU vertexBuffer if dirty﻿    // Vulkan draw﻿﻿              << mesh.vertices.size() << " vertices\n";﻿    // Example draw logic:﻿    // vkCmdBindPipeline(...)﻿    // vkCmdBindVertexBuffers(...)﻿    // vkCmdDraw(...)﻿}﻿extern "C" void generate\_chunk\_mesh(float chunk\_x, float chunk\_y, float chunk\_z) {﻿    ChunkMesh mesh;﻿    mesh.vertices.clear();﻿    // Simple voxel cube mesh for demo (1 block)﻿    for(int x=0;x<16;x++){﻿        for(int y=0;y<16;y++){﻿            for(int z=0;z<16;z++){﻿                BlockVertex v;﻿                v.pos = {chunk\_x+x, chunk\_y+y, chunk\_z+z};﻿                v.color[0]=0.5f; v.color[1]=0.8f; v.color[2]=0.5f;﻿                mesh.vertices.push\_back(v);﻿            }﻿        }﻿    }﻿    loadedChunks.push\_back(mesh);﻿﻿}﻿public class JarockGame {﻿    static { System.loadLibrary("jarock\_chunk"); }﻿    public native void createChunk(int x, int y, int z);﻿    public native void updateChunk(int index);﻿    public native void generateChunkMesh(float x, float y, float z);﻿    public native void renderChunk(int index);﻿    public void onPlayerLookChunk(int chunkX,int chunkY,int chunkZ, int index){﻿        createChunk(chunkX,chunkY,chunkZ);   // Rust backend﻿        updateChunk(index);                   // Rust 10%﻿        generateChunkMesh(chunkX,chunkY,chunkZ); // C++ 90%﻿        renderChunk(index);                   // Vulkan draw﻿    }﻿}﻿import java.util.\*;﻿public class JarockGame {﻿    static { System.loadLibrary("jarock\_chunk"); }﻿    public native void createChunk(int x,int y,int z);﻿    public native void updateChunk(int index);﻿    public native void generateChunkMesh(float x,float y,float z);﻿    public native void renderChunk(int index);﻿    // Sparse map: key = chunk coord (x,y,z) as string﻿    Map<String,Integer> loadedChunks = new HashMap<>();﻿    int chunkCounter = 0;﻿    // Player position + view distance﻿    float playerX, playerY, playerZ;﻿    int viewDistance = 3; // chunks﻿    private String key(int x,int y,int z){ return x+","+y+","+z; }﻿    // Call when player moves/look﻿    public void updateVisibleChunks(){﻿        Set<String> newVisible = new HashSet<>();﻿        int px = (int)Math.floor(playerX/16);﻿        int py = (int)Math.floor(playerY/16);﻿        int pz = (int)Math.floor(playerZ/16);﻿        // Determine visible chunks around player﻿        for(int x=px-viewDistance;x<=px+viewDistance;x++){﻿            for(int y=py-viewDistance;y<=py+viewDistance;y++){﻿                for(int z=pz-viewDistance;z<=pz+viewDistance;z++){﻿                    String k = key(x,y,z);﻿                    newVisible.add(k);﻿                    if(!loadedChunks.containsKey(k)){﻿                        createChunk(x,y,z);        // Rust backend﻿                        generateChunkMesh(x,y,z);  // C++ 90%﻿                        loadedChunks.put(k,chunkCounter++);﻿                    }﻿                }﻿            }﻿        }﻿        // Remove chunks outside view to save RAM﻿        Iterator<Map.Entry<String,Integer>> it = loadedChunks.entrySet().iterator();﻿        while(it.hasNext()){﻿            Map.Entry<String,Integer> e = it.next();﻿            if(!newVisible.contains(e.getKey())){﻿                // Optionally call Rust/C++ to free buffers﻿                it.remove();﻿            }﻿        }﻿    }﻿    // Render visible chunks only﻿    public void renderVisibleChunks(){﻿        for(int index: loadedChunks.values()){﻿            updateChunk(index);   // Rust update 10%﻿            renderChunk(index);   // C++ render﻿        }﻿    }﻿}﻿#[derive(Clone,Copy)]﻿pub struct Block{ pub id:u8; }﻿pub struct Chunk{ pub x:i32; pub y:i32; pub z:i32; pub blocks: Vec<Block>; }﻿static mut CHUNKS: Vec<Chunk> = Vec::new();﻿// Create chunk﻿#[no\_mangle]﻿pub extern "C" fn create\_chunk(x:i32,y:i32,z:i32){﻿    let mut blocks = vec![Block{id:0};16\*16\*16];﻿    for i in 0..16{ for j in 0..16{ for k in 0..16{﻿        if j==0{ blocks[i+16\*(j+16\*k)].id =1; }﻿    }}}﻿    unsafe { CHUNKS.push(Chunk{x,y,z,blocks}); }﻿}﻿// Update only visible chunk by index﻿#[no\_mangle]﻿pub extern "C" fn update\_chunk(index:usize){﻿    unsafe{﻿        if let Some(chunk)=CHUNKS.get\_mut(index){﻿            // Example: 10% CPU load - block growth﻿            for block in chunk.blocks.iter\_mut(){﻿                if block.id==0{ block.id=0; } // placeholder﻿            }﻿        }﻿    }﻿}﻿#include <vector>﻿#include <iostream>﻿struct Vec3{ float x,y,z; };﻿struct BlockVertex{ Vec3 pos; float color[3]; };﻿struct ChunkMesh{ std::vector<BlockVertex> vertices; };﻿std::vector<ChunkMesh> loadedChunks;﻿extern "C" void generateChunkMesh(float chunk\_x,float chunk\_y,float chunk\_z){﻿    ChunkMesh mesh;﻿    mesh.vertices.clear();﻿    for(int x=0;x<16;x++){﻿        for(int y=0;y<16;y++){﻿            for(int z=0;z<16;z++){﻿                BlockVertex v;﻿                v.pos = {chunk\_x+x,chunk\_y+y,chunk\_z+z};﻿                v.color[0]=0.5f; v.color[1]=0.8f; v.color[2]=0.5f;﻿                mesh.vertices.push\_back(v);﻿            }﻿        }﻿    }﻿    loadedChunks.push\_back(mesh);﻿﻿}﻿extern "C" void renderChunk(int chunk\_index){﻿    if(chunk\_index>=loadedChunks.size()) return;﻿    ChunkMesh &mesh=loadedChunks[chunk\_index];﻿﻿    // Vulkan draw﻿}﻿use std::collections::HashMap;﻿use std::time::{Instant, Duration};﻿// ----- Mob -----﻿#[derive(Clone)]﻿pub struct Mob {﻿    pub id: u32,﻿    pub x: f32, pub y: f32, pub z: f32,﻿    pub alive: bool,﻿}﻿// ----- Item -----﻿#[derive(Clone)]﻿pub struct Item {﻿    pub id: u32,﻿    pub x: f32, pub y: f32, pub z: f32,﻿    pub spawn\_time: Instant,﻿    pub picked: bool,﻿}﻿static mut MOBS: Option<HashMap<u32, Mob>> = None;﻿static mut ITEMS: Option<HashMap<u32, Item>> = None;﻿#[no\_mangle]﻿pub extern "C" fn init\_game() {﻿    unsafe { ﻿        MOBS = Some(HashMap::new());﻿        ITEMS = Some(HashMap::new());﻿    }﻿}﻿// Spawn mob﻿#[no\_mangle]﻿pub extern "C" fn spawn\_mob(id:u32, x:f32, y:f32, z:f32) {﻿    let mob = Mob { id, x, y, z, alive:true };﻿    unsafe { MOBS.as\_mut().unwrap().insert(id, mob); }﻿}﻿// Spawn item﻿#[no\_mangle]﻿pub extern "C" fn spawn\_item(id:u32, x:f32, y:f32, z:f32) {﻿    let item = Item { id, x, y, z, spawn\_time: Instant::now(), picked:false };﻿    unsafe { ITEMS.as\_mut().unwrap().insert(id, item); }﻿}﻿// Mob nhận sát thương => chết﻿#[no\_mangle]﻿pub extern "C" fn mob\_take\_damage(id:u32) {﻿    unsafe {﻿        if let Some(mobs) = &mut MOBS {﻿            if let Some(mob) = mobs.get\_mut(&id) {﻿                mob.alive = false; // chết﻿            }﻿        }﻿    }﻿}﻿// Update items: despawn sau 5 phút﻿#[no\_mangle]﻿pub extern "C" fn update\_items() {﻿    unsafe {﻿        if let Some(items) = &mut ITEMS {﻿            let now = Instant::now();﻿            items.retain(|\_,item| !item.picked && now.duration\_since(item.spawn\_time) < Duration::from\_secs(300));﻿        }﻿    }﻿}﻿// Remove dead mobs﻿#[no\_mangle]﻿pub extern "C" fn cleanup\_mobs() {﻿    unsafe {﻿        if let Some(mobs) = &mut MOBS {﻿            mobs.retain(|\_,mob| mob.alive);﻿        }﻿    }﻿}﻿#include <vector>﻿#include <iostream>﻿struct Vec3 { float x,y,z; };﻿struct MobMesh { Vec3 pos; bool alive; };﻿struct ItemMesh { Vec3 pos; bool picked; };﻿std::vector<MobMesh> mobMeshes;﻿std::vector<ItemMesh> itemMeshes;﻿// Render mobs﻿extern "C" void render\_mobs(){﻿    for(auto it=mobMeshes.begin(); it!=mobMeshes.end(); ){﻿        if(!it->alive){ it = mobMeshes.erase(it); continue; }﻿﻿        ++it;﻿    }﻿}﻿// Render items﻿extern "C" void render\_items(){﻿    for(auto it=itemMeshes.begin(); it!=itemMeshes.end(); ){﻿        if(it->picked){ it = itemMeshes.erase(it); continue; }﻿﻿        ++it;﻿    }﻿}﻿import java.util.\*;﻿public class JarockGame {﻿    static { System.loadLibrary("jarock\_game"); }﻿    public native void initGame();﻿    public native void spawnMob(int id,float x,float y,float z);﻿    public native void spawnItem(int id,float x,float y,float z);﻿    public native void mobTakeDamage(int id);﻿    public native void updateItems();﻿    public native void cleanupMobs();﻿    public native void renderMobs();﻿    public native void renderItems();﻿    List<Integer> mobs = new ArrayList<>();﻿    List<Integer> items = new ArrayList<>();﻿    public void gameLoop(){﻿        while(true){﻿            // --- update logic ---﻿            updateItems();       // despawn items sau 5 phút﻿            cleanupMobs();       // remove dead mobs﻿            // --- render ---﻿            renderMobs();﻿            renderItems();﻿            try{ Thread.sleep(50); } catch(Exception e){}﻿        }﻿    }﻿    // Ví dụ: player đánh mob﻿    public void playerAttack(int mobId){﻿        mobTakeDamage(mobId); // mob sẽ chết ngay﻿    }﻿}﻿import java.util.\*;﻿import java.util.concurrent.\*;﻿public class JarockGame {﻿    static { System.loadLibrary("jarock\_game"); }﻿    // --- Native calls ---﻿    public native void initGame();﻿    public native void spawnMob(int id,float x,float y,float z);﻿    public native void spawnItem(int id,float x,float y,float z);﻿    public native void mobTakeDamage(int id);﻿    public native void updateItems();﻿    public native void cleanupMobs();﻿    public native void renderMobs();﻿    public native void renderItems();﻿    public native void loadChunkAsync(int x,int y,int z);﻿    // --- Game state ---﻿    List<Integer> mobs = new ArrayList<>();﻿    List<Integer> items = new ArrayList<>();﻿    float playerX, playerY, playerZ;﻿    ExecutorService executor = Executors.newFixedThreadPool(4); // threads for async tasks﻿    // --- Input handling ---﻿    Set<Integer> keysPressed = ConcurrentHashMap.newKeySet();﻿    public void handleInput(){﻿        // WASD movement demo﻿        if(keysPressed.contains(87)) playerZ+=0.1f; // W﻿        if(keysPressed.contains(83)) playerZ-=0.1f; // S﻿        if(keysPressed.contains(65)) playerX-=0.1f; // A﻿        if(keysPressed.contains(68)) playerX+=0.1f; // D﻿    }﻿    // --- Asynchronous chunk loading ---﻿    public void loadVisibleChunks(int viewDistance){﻿        int px = (int)Math.floor(playerX/16);﻿        int py = (int)Math.floor(playerY/16);﻿        int pz = (int)Math.floor(playerZ/16);﻿        for(int x=px-viewDistance;x<=px+viewDistance;x++){﻿            for(int y=py-viewDistance;y<=py+viewDistance;y++){﻿                for(int z=pz-viewDistance;z<=pz+viewDistance;z++){﻿                    int finalX=x, finalY=y, finalZ=z;﻿                    executor.submit(()->{﻿                        loadChunkAsync(finalX, finalY, finalZ); // Rust/C++ load chunk﻿                    });﻿                }﻿            }﻿        }﻿    }﻿    // --- Game loop ---﻿    public void startGameLoop(){﻿        new Thread(()->{﻿            while(true){﻿                handleInput();       // player input﻿                // --- Update logic ---﻿                updateItems();       // despawn items﻿                cleanupMobs();       // remove dead mobs﻿                // --- Render ---﻿                renderMobs();﻿                renderItems();﻿                try{ Thread.sleep(16); } catch(Exception e){}﻿            }﻿        }).start();﻿    }﻿    // --- Network sync thread ---﻿    public void startNetworkThread(){﻿        new Thread(()->{﻿            while(true){﻿                // TODO: send player position / receive other players﻿                try{ Thread.sleep(50); } catch(Exception e){}﻿            }﻿        }).start();﻿    }﻿    // --- Input registration ---﻿    public void keyPressed(int keyCode){ keysPressed.add(keyCode); }﻿    public void keyReleased(int keyCode){ keysPressed.remove(keyCode); }﻿    // --- Entry point ---﻿    public static void main(String[] args){﻿        JarockGame game = new JarockGame();﻿        game.initGame();﻿        // Spawn demo mob/item﻿        game.spawnMob(1,0,10,0);﻿        game.spawnItem(1,5,0,5);﻿        // Start threads﻿        game.startGameLoop();﻿        game.startNetworkThread();﻿        game.loadVisibleChunks(3); // initial chunk load﻿    }﻿}﻿use std::collections::HashMap;﻿use std::time::{Instant, Duration};﻿#[derive(Clone)]﻿pub struct Mob {﻿    pub id: u32,﻿    pub x: f32, pub y: f32, pub z: f32,﻿    pub vx: f32, pub vy: f32, pub vz: f32,﻿    pub alive: bool,﻿    pub health: f32,﻿}﻿#[derive(Clone)]﻿pub struct Item {﻿    pub id: u32,﻿    pub x: f32, pub y: f32, pub z: f32,﻿    pub spawn\_time: Instant,﻿    pub picked: bool,﻿}﻿static mut MOBS: Option<HashMap<u32, Mob>> = None;﻿static mut ITEMS: Option<HashMap<u32, Item>> = None;﻿#[no\_mangle]﻿pub extern "C" fn init\_game() {﻿    unsafe { ﻿        MOBS = Some(HashMap::new());﻿        ITEMS = Some(HashMap::new());﻿    }﻿}﻿#[no\_mangle]﻿pub extern "C" fn spawn\_mob(id:u32, x:f32, y:f32, z:f32){﻿    let mob = Mob{id, x,y,z, vx:0.0, vy:0.0, vz:0.0, alive:true, health:100.0};﻿    unsafe { MOBS.as\_mut().unwrap().insert(id, mob); }﻿}﻿#[no\_mangle]﻿pub extern "C" fn spawn\_item(id:u32, x:f32, y:f32, z:f32){﻿    let item = Item{id, x,y,z, spawn\_time:Instant::now(), picked:false};﻿    unsafe { ITEMS.as\_mut().unwrap().insert(id,item); }﻿}﻿// Mob nhận sát thương bất kỳ -> chết﻿#[no\_mangle]﻿pub extern "C" fn mob\_take\_damage(id:u32, dmg:f32){﻿    unsafe {﻿        if let Some(mobs) = &mut MOBS {﻿            if let Some(mob) = mobs.get\_mut(&id){﻿                mob.health -= dmg;﻿                if mob.health <= 0.0 { mob.alive = false; }﻿            }﻿        }﻿    }﻿}﻿// Update AI + physics + collision﻿#[no\_mangle]﻿pub extern "C" fn update\_entities(player\_positions:\*const (f32,f32,f32), player\_count:usize){﻿    unsafe {﻿        if let Some(mobs) = &mut MOBS {﻿            for mob in mobs.values\_mut(){﻿                if !mob.alive { continue; }﻿                // Gravity﻿                mob.vy -= 0.01;﻿                mob.y += mob.vy;﻿                if mob.y < 0.0 { mob.y=0.0; mob.vy=0.0; }﻿                // Follow nearest player﻿                if player\_count>0{﻿                    let mut nearest=0; let mut min\_dist=f32::MAX;﻿                    for i in 0..player\_count{﻿                        let (px,py,pz) = \*player\_positions.add(i);﻿                        let dist2=(mob.x-px).powi(2)+(mob.y-py).powi(2)+(mob.z-pz).powi(2);﻿                        if dist2<min\_dist { min\_dist=dist2; nearest=i; }﻿                    }﻿                    let (tx,ty,tz) = \*player\_positions.add(nearest);﻿                    let dx = tx-mob.x; let dz = tz-mob.z;﻿                    let len = (dx\*dx+dz\*dz).sqrt();﻿                    if len>0.0{﻿                        mob.vx = dx/len\*0.05;﻿                        mob.vz = dz/len\*0.05;﻿                    }﻿                }﻿                // Simple collision with other mobs﻿                for other in mobs.values(){﻿                    if other.id != mob.id && other.alive {﻿                        let dx = mob.x-other.x;﻿                        let dz = mob.z-other.z;﻿                        let dist2 = dx\*dx+dz\*dz;﻿                        if dist2 < 1.0 { mob.vx += dx\*0.01; mob.vz += dz\*0.01; }﻿                    }﻿                }﻿                mob.x += mob.vx;﻿                mob.z += mob.vz;﻿            }﻿        }﻿        // Item despawn after 5 min﻿        if let Some(items) = &mut ITEMS {﻿            let now = Instant::now();﻿            items.retain(|\_,item| !item.picked && now.duration\_since(item.spawn\_time) < Duration::from\_secs(300));﻿        }﻿        // Remove dead mobs﻿        if let Some(mobs) = &mut MOBS { mobs.retain(|\_,mob| mob.alive); }﻿    }﻿}﻿#include <vulkan/vulkan.h>﻿#include <vector>﻿#include <iostream>﻿VkInstance instance;﻿VkDevice device;﻿VkQueue graphicsQueue;﻿VkSwapchainKHR swapChain;﻿VkRenderPass renderPass;﻿VkPipelineLayout pipelineLayout;﻿VkPipeline graphicsPipeline;﻿struct Vec3{ float x,y,z; };﻿struct MobMesh{ Vec3 pos; };﻿std::vector<MobMesh> mobsToRender;﻿extern "C" void init\_vulkan() {﻿    // TODO: full Vulkan initialization: instance, physical device, logical device﻿    std::cout<<"Vulkan initialized\n";﻿    // RenderPass + Pipeline setup placeholder﻿    // Swapchain, Framebuffers, Shaders...﻿}﻿extern "C" void update\_mob\_render(float\* mob\_positions, int mob\_count){﻿    mobsToRender.clear();﻿    for(int i=0;i<mob\_count;i++){﻿        mobsToRender.push\_back({mob\_positions[i\*3], mob\_positions[i\*3+1], mob\_positions[i\*3+2]});﻿    }﻿}﻿extern "C" void render\_vulkan\_frame(){﻿    // TODO: Bind pipeline, vertex buffer, draw mobs﻿    for(auto &mob : mobsToRender){﻿﻿    }﻿}﻿import java.util.\*;﻿import java.util.concurrent.\*;﻿public class JarockGame {﻿    static { System.loadLibrary("jarock\_game"); }﻿    public native void initGame();﻿    public native void spawnMob(int id,float x,float y,float z);﻿    public native void spawnItem(int id,float x,float y,float z);﻿    public native void mobTakeDamage(int id,float dmg);﻿    public native void updateEntities(float[] playerPositions,int playerCount);﻿    public native void updateMobRender(float[] mobPositions,int mobCount);﻿    public native void renderFrame();﻿    List<float[]> players = new ArrayList<>();﻿    ExecutorService executor = Executors.newFixedThreadPool(2);﻿    public void gameLoop(){﻿        new Thread(()->{﻿            while(true){﻿                float[] playerPosArray = flattenPlayerPositions();﻿                updateEntities(playerPosArray, players.size());﻿                // Gather mob positions to render﻿                float[] mobPositions = getMobPositions(); // call Rust to get mob positions﻿                updateMobRender(mobPositions, mobPositions.length/3);﻿                // Render﻿                renderFrame();﻿                try{ Thread.sleep(16); } catch(Exception e){}﻿            }﻿        }).start();﻿    }﻿    public float[] flattenPlayerPositions(){﻿        float[] arr = new float[players.size()\*3];﻿        for(int i=0;i<players.size();i++){﻿            float[] p=players.get(i);﻿            arr[i\*3]=p[0]; arr[i\*3+1]=p[1]; arr[i\*3+2]=p[2];﻿        }﻿        return arr;﻿    }﻿    public float[] getMobPositions(){﻿        // Placeholder: in reality, call Rust FFI to get all mob positions﻿        return new float[]{0f,0f,0f};﻿    }﻿}﻿cmake\_minimum\_required(VERSION 3.21)﻿project(jarock\_vk)﻿set(CMAKE\_CXX\_STANDARD 17)﻿find\_package(Vulkan REQUIRED)﻿add\_subdirectory(external/glfw) # dùng GLFW cho surface﻿add\_library(jarock\_vk STATIC renderer.cpp renderer.h shaders.spv.h)﻿target\_include\_directories(jarock\_vk PRIVATE ${Vulkan\_INCLUDE\_DIR})﻿target\_link\_libraries(jarock\_vk PRIVATE glfw Vulkan::Vulkan)﻿#pragma once﻿#include <cstdint>﻿extern "C" {﻿  bool vk\_init(int width, int height, const char\* title);﻿  void vk\_upload\_instances(const float\* xyz, int count); // xyz packed﻿  void vk\_draw\_frame();﻿  void vk\_shutdown();﻿}