public class Main {

public static void main(String[] args) throws Exception {

Game game = new Game();

game.run();

}

}

public class Game {

private long window;

private int width = 1280;

private int height = 720;

private WorldManager worldManager;

private Camera camera;

private ShaderProgram shaderProgram;

private ShaderProgram selectorShader;

private Cube cubePreview;

//Fuente

private TextRenderer textRenderer;

//TEXTURAAAAS

private Texture grassT;

private Texture dirtT;

private Texture stoneT;

//BLOQUES DE STEVE

private final Block[] availableBlocks = {Block.GRASS, Block.DIRT, Block.STONE};

private int selectedBlockIndex = 0;

private Block selectedBlock = availableBlocks[selectedBlockIndex];

private float mouseSensitivity = 0.1f;

private float movementSpeed = 5f;

private float floatingCheckTimer = 0;

public static float gravityDelayTime = 1.0f;

public static final float GRAVITY\_COUNTDOWN\_TIME = 0.25f;

private final List<Vector3i> fallingBlocks = new ArrayList<>();

//Click x default

private boolean leftClickPressed = false;

private boolean rightClickPressed = false;

private boolean leftClickHeld = false;

private float destroyCountDown = 0f;

private float placeCountDown = 0f;

private float fallCountDown = 0f;

private final float DESTROY\_COUNTDOWN\_TIME = 0.25f;

private final float PLACE\_COUNTDOWN\_TIME = 0.25f;

private final float FALL\_COUNTDOWN\_TIME = 0.20f;

private BlockSelector blockSelector;

private ChunkLoaderThread chunkLoader;

// Para movimiento del mouse

private double lastMouseX = width / 2.0;

private double lastMouseY = height / 2.0;

private boolean firstMouse = true;

private float scrollOffsetY = 0;

private Vector3f velocity = new Vector3f(0, 0, 0);

private final float gravity = -30;

private final float jumpForce = 10f;

private boolean isOnGround = false;

public void run() throws Exception {

init();

loop();

cleanup();

}

private void init() throws Exception {

GLFWErrorCallback.createPrint(System.err).set();

if (!glfwInit()) {

throw new IllegalStateException("No se pudo inicializar GLFW");

}

glfwWindowHint(GLFW\_VISIBLE, GLFW\_FALSE);

glfwWindowHint(GLFW\_RESIZABLE, GLFW\_TRUE);

window = GLFW.glfwCreateWindow(width, height, "Minecraft Clone", NULL, NULL);

if (window == NULL) {

throw new RuntimeException("No se pudo crear la ventana GLFW");

}

glfwMakeContextCurrent(window);

glfwSwapInterval(1);

glfwShowWindow(window);

glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);

GL.createCapabilities();

glEnable(GL\_DEPTH\_TEST);

glClearColor(0.53f, 0.8f, 0.92f, 1.0f);

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glViewport(0, 0, width, height);

glfwSetScrollCallback(window, (win, xoffset, yoffset) -> {

if (yoffset > 0) {

selectedBlockIndex = (selectedBlockIndex + 1) % availableBlocks.length;

} else if (yoffset < 0) {

selectedBlockIndex = (selectedBlockIndex - 1 + availableBlocks.length) % availableBlocks.length;

}

selectedBlock = availableBlocks[selectedBlockIndex];

textRenderer.updateText(selectedBlock.getType().name());

System.out.println("Bloque seleccionado (scroll): " + selectedBlock.getType());

});

camera = new Camera();

blockSelector = new BlockSelector();

shaderProgram = new ShaderProgram("res/shaders/vertex.glsl", "res/shaders/fragment.glsl");

selectorShader = new ShaderProgram("res/shaders/selector\_vertex.glsl", "res/shaders/selector\_fragment.glsl");

grassT = new Texture("res/textures/grass\_texture.png");

dirtT = new Texture("res/textures/dirt\_texture.png");

stoneT = new Texture("res/textures/stone\_texture.png");

cubePreview = new Cube();

// Font font = new Font("Arial", Font.BOLD, 32);

Font font = null;

try {

font = Font.createFont(Font.TRUETYPE\_FONT, new File("res/fonts/MinecraftRegular-Bmg3.otf")).deriveFont(32f);

GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();

ge.registerFont(font);

} catch (Exception e) {

e.printStackTrace();

font = new Font("Arial", Font.BOLD, 32); // fallback

}

textRenderer = new TextRenderer(selectedBlock.getType().toString(), font, Color.WHITE);

// 1) Crear el WorldManager

worldManager = new WorldManager(camera, shaderProgram, grassT, dirtT, stoneT);

camera.setPosition(8f, 5.1f, 8f);

isOnGround = true;

//2) \*\*Crear SINCRÓNICAMENTE el chunk base (0,0,0) para que nunca caigamos al vacío\*\*:

Chunk baseChunk = new Chunk(0, 0, 0);

//Confirmar si existe ya:

File file = new File(worldFolder, "0,0,0.dat");

if (file.exists()) {

try (DataInputStream in = new DataInputStream(new FileInputStream(file))) {

// Inicializar todo con AIR primero para evitar nulos

for (int x = 0; x < Chunk.SIZEx; x++) {

for (int y = terrainHeight; y < Chunk.SIZEy; y++) {

for (int z = 0; z < Chunk.SIZEz; z++) {

baseChunk.setBlock(x, y, z, Block.AIR);

}

}

}

int layerCount = in.readInt(); // cuántas capas hay

for (int i = 0; i < layerCount; i++) {

int y = in.readInt(); // índice de la capa

for (int x = 0; x < Chunk.SIZEx; x++) {

for (int z = 0; z < Chunk.SIZEz; z++) {

int id = in.readInt();

Block b = Block.getBlockById(id);

baseChunk.setBlock(x, y, z, b);

}

}

}

baseChunk.buildMesh(); // crea VAO/VBO para dibujar

String key = baseChunk.getKey();

worldManager.chunks.put(key, baseChunk); // mete el chunk base en el mapa de chunks

}

} else { //no existe, lo hacemos nuevo

baseChunk.generate();

baseChunk.buildMesh(); // crea VAO/VBO para dibujar

String key = baseChunk.getKey();

worldManager.chunks.put(key, baseChunk); // mete el chunk base en el mapa de chunks

}

// 4) Ahora sí creo y arranco el hilo para cargar el resto de chunks en segundo plano

chunkLoader = new ChunkLoaderThread(worldManager);

worldManager.setChunkLoader(chunkLoader);

MeshBuilderThread meshThread = new MeshBuilderThread(worldManager);

worldManager.setMeshBuilder(meshThread);

chunkLoader.start();

worldManager.startThreads();

}

private void detectFloatingBlocksInCurrentChunk() {

Vector3f camPos = camera.getPosition();

int chunkX = Math.floorDiv((int) camPos.x, Chunk.SIZEx);

int chunkY = Math.floorDiv((int) camPos.y, Chunk.SIZEy);

int chunkZ = Math.floorDiv((int) camPos.z, Chunk.SIZEz);

Chunk chunk = worldManager.getChunk(chunkX, chunkY, chunkZ);

if (chunk == null) {

return;

}

for (int x = 0; x < Chunk.SIZEx; x++) {

for (int y = 0; y < Chunk.SIZEy; y++) {

for (int z = 0; z < Chunk.SIZEz; z++) {

Block blockId = chunk.getBlock(x, y, z);

if (blockId == AIR) {

continue;

}

int worldX = chunkX \* Chunk.SIZEx + x;

int worldY = chunkY \* Chunk.SIZEy + y;

int worldZ = chunkZ \* Chunk.SIZEz + z;

if (!worldManager.hasSupport(worldX, worldY, worldZ)) {

Vector3i pos = new Vector3i(worldX, worldY, worldZ);

if (!fallingBlocks.contains(pos)) {

fallingBlocks.add(pos);

}

}

}

}

}

}

private void applyGravity(float deltaTime) {

if (gravityDelayTime > 0) {

gravityDelayTime -= deltaTime;

return;

}

Block check = worldManager.getBlockIfLoader((int) camera.getPosition().x, (int) camera.getPosition().y, (int) camera.getPosition().z);

if (check != null) {

if (check.isSolid()) {

isOnGround = true;

velocity.y = 0;

// Ajustamos para evitar flotar o atravesar el suelo

float alignedY = (float) Math.floor(camera.getPosition().y);

camera.setPosition(camera.getPosition().x, alignedY, camera.getPosition().z);

return;

}

}

if (!isOnGround) {

velocity.y += gravity \* deltaTime;

}

Vector3f nextPos = new Vector3f(camera.getPosition());

nextPos.y += velocity.y \* deltaTime;

if (collidesWithWorld(nextPos)) {

if (velocity.y < 0) {

isOnGround = true;

velocity.y = 0;

// Ajustamos para evitar flotar o atravesar el suelo

float alignedY = (float) Math.floor(camera.getPosition().y);

camera.setPosition(camera.getPosition().x, alignedY, camera.getPosition().z);

}

} else {

camera.setPosition(camera.getPosition().x, nextPos.y, camera.getPosition().z);

isOnGround = false;

}

}

private void loop() throws Exception {

float lastTime = (float) glfwGetTime();

while (!glfwWindowShouldClose(window)) {

float currentTime = (float) glfwGetTime();

float deltaTime = currentTime - lastTime;

lastTime = currentTime;

glfwPollEvents();

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

handleMouse();

handleKeyboard(deltaTime);

applyGravity(deltaTime);

RaycastResult selectedblock = RaycastUtils.raycast(worldManager, camera.getEyePosition(), camera.getFront(), 5.0f);

if (selectedblock != null) {

blockSelector.render(selectedblock.blockPos, camera.getProjectionMatrix(70, (float) width / height, 0.01f, 1000f),

camera.getViewMatrix(), selectorShader);

System.out.println("Apuntas al bloque: " + selectedblock.blockPos + " cara: " + selectedblock.faceNormal);

Vector3f blockPos = selectedblock.blockPos;

Vector3f normal = selectedblock.faceNormal;

destroyCountDown -= deltaTime;

placeCountDown -= deltaTime;

// Clic izquierdo = destruir bloque

boolean isLeftPressed = GLFW.glfwGetMouseButton(window, GLFW.GLFW\_MOUSE\_BUTTON\_LEFT) == GLFW\_PRESS;

if (destroyCountDown <= 0 && isLeftPressed) {

worldManager.setBlockIfChunkExists((int) blockPos.x, (int) blockPos.y, (int) blockPos.z, Block.AIR);// 0 = aire

destroyCountDown = DESTROY\_COUNTDOWN\_TIME;

}

leftClickPressed = isLeftPressed;

// Clic derecho = colocar bloque al lado opuesto de la cara

boolean isRightPressed = GLFW.glfwGetMouseButton(window, GLFW.GLFW\_MOUSE\_BUTTON\_RIGHT) == GLFW\_PRESS;

if (placeCountDown <= 0 && isRightPressed) {

int x = (int) (blockPos.x + normal.x);

int y = (int) (blockPos.y + normal.y);

int z = (int) (blockPos.z + normal.z);

Vector3f placement = new Vector3f(x, y, z);

Vector3f playerPos = camera.getPosition();

float playerSize = Camera.PLAYER\_WIDTH; // anchura del jugador

float playerHeight = Camera.PLAYER\_HEIGHT;

boolean insidePlayer = placement.x + 1 > playerPos.x - playerSize / 2f

&& placement.x < playerPos.x + playerSize / 2f

&& placement.y + 1 > playerPos.y

&& placement.y < playerPos.y + playerHeight

&& placement.z + 1 > playerPos.z - playerSize / 2f

&& placement.z < playerPos.z + playerSize / 2f;

//gravityDelayTime = GRAVITY\_COUNTDOWN\_TIME;

if (!insidePlayer) {

worldManager.setBlockIfChunkExists(x, y, z, selectedBlock);

}

placeCountDown = PLACE\_COUNTDOWN\_TIME;

}

rightClickPressed = isRightPressed;

floatingCheckTimer += deltaTime;

if (floatingCheckTimer < 0.5f) {

detectFloatingBlocksInCurrentChunk();

floatingCheckTimer = 0;

}

}

Iterator<Vector3i> it = fallingBlocks.iterator();

while (it.hasNext()) {

Vector3i pos = it.next();

Block bellow = worldManager.getBlockIfLoader(pos.x, pos.y - 1, pos.z);

Block act = worldManager.getBlockIfLoader(pos.x, pos.y, pos.z);

fallCountDown -= deltaTime;

if (fallCountDown <= 0) {

if (bellow == AIR) {

worldManager.setBlock(pos.x, pos.y, pos.z, AIR);

worldManager.setBlock(pos.x, pos.y - 1, pos.z, act);

pos.y -= 1;

} else {

it.remove();

}

fallCountDown = FALL\_COUNTDOWN\_TIME;

}

}

//Integrar chunks

worldManager.integrateLoadedChunks();

//UpdateChunks

worldManager.updateChunks(camera.getPosition());

// Renderizar chunks

worldManager.render();

//Bloque seleccionado en pantalla

glMatrixMode(GL\_PROJECTION);

glPushMatrix();

glLoadIdentity();

glOrtho(0, width, height, 0, -1, 1); // coordenadas en píxeles

glMatrixMode(GL\_MODELVIEW);

glPushMatrix();

glLoadIdentity();

textRenderer.render(20, 20); // dibuja en pantalla

glPopMatrix();

glMatrixMode(GL\_PROJECTION);

glPopMatrix();

glMatrixMode(GL\_MODELVIEW);

//Cubo seleccionado imagen

Matrix4f projection = new Matrix4f().perspective((float) Math.toRadians(45), (float) width / height, 0.1f, 100f);

Matrix4f view = new Matrix4f()

.translate(0, 0, -3f); // Cámara que apunta a (0,0,0) desde Z = -3

//cube en pantalla

cubePreview.setRotation(new Vector3f(30f, currentTime \* 50f, 0f)); // Gira suavemente

cubePreview.render(selectedBlock.getTexture(), projection,

view, shaderProgram);

//Mira central

glDisable(GL\_DEPTH\_TEST);

glBegin(GL\_LINES);

glColor3f(1, 1, 1); // blanco

// Línea horizontal

glVertex2f(width / 2f - 5, height / 2f);

glVertex2f(width / 2f + 5, height / 2f);

// Línea vertical

glVertex2f(width / 2f, height / 2f - 5);

glVertex2f(width / 2f, height / 2f + 5);

glEnd();

glEnable(GL\_DEPTH\_TEST);

glfwSwapBuffers(window);

System.out.println("IsOnGround: " + isOnGround);

}

}

private void handleMouse() {

try (MemoryStack stack = MemoryStack.stackPush()) {

DoubleBuffer xpos = stack.mallocDouble(1);

DoubleBuffer ypos = stack.mallocDouble(1);

glfwGetCursorPos(window, xpos, ypos);

double x = xpos.get(0);

double y = ypos.get(0);

if (firstMouse) {

lastMouseX = x;

lastMouseY = y;

firstMouse = false;

}

float xoffset = (float) (x - lastMouseX);

float yoffset = (float) (y - lastMouseY);

lastMouseX = x;

lastMouseY = y;

camera.processMouseMovement(xoffset, yoffset, mouseSensitivity);

if (scrollOffsetY != 0) {

int direction = (int) Math.signum(scrollOffsetY);

scrollOffsetY = 0; // Reiniciamos tras usarlo

selectedBlockIndex = (selectedBlockIndex + direction + availableBlocks.length) % availableBlocks.length;

selectedBlock = availableBlocks[selectedBlockIndex];

System.out.println("Bloque seleccionado: " + selectedBlock.getType());

}

}

}

private void handleKeyboard(float deltaTime) {

if (gravityDelayTime > 0) {

gravityDelayTime -= deltaTime;

return;

}

float speed = movementSpeed \* deltaTime;

Vector3f foward = new Vector3f(camera.getFront()).setComponent(1, 0).normalize();

Vector3f right = new Vector3f(camera.getRight()).setComponent(1, 0).normalize();

if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS) {

move(foward, speed);

}

if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS) {

move(foward.negate(), speed);

}

if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS) {

move(right.negate(), speed);

}

if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS) {

move(right, speed);

}

if (glfwGetKey(window, GLFW\_KEY\_SPACE) == GLFW\_PRESS && isOnGround) {

velocity.y = jumpForce;

isOnGround = false;

}

if (glfwGetKey(window, GLFW\_KEY\_LEFT\_SHIFT) == GLFW\_PRESS) {

move(new Vector3f(0, -1, 0), speed);

}

}

private void move(Vector3f offset, float speed) {

Vector3f proposedPos = new Vector3f(camera.getPosition()).add(offset.mul(speed));

if (!collidesWithWorld(proposedPos)) {

camera.setPosition(proposedPos.x, proposedPos.y, proposedPos.z);

}

}

private boolean collidesWithWorld(Vector3f pos) {

Vector3f min = new Vector3f(

pos.x - Camera.PLAYER\_WIDTH / 2f,

pos.y,

pos.z - Camera.PLAYER\_WIDTH / 2f

);

Vector3f max = new Vector3f(

pos.x + Camera.PLAYER\_WIDTH / 2f,

pos.y + Camera.PLAYER\_HEIGHT,

pos.z + Camera.PLAYER\_WIDTH / 2f

);

// Debug

System.out.println("Colisión check de AABB entre " + min + " y " + max);

for (int x = (int) Math.floor(min.x); x <= Math.floor(max.x); x++) {

for (int y = (int) Math.floor(min.y); y <= Math.floor(max.y); y++) {

for (int z = (int) Math.floor(min.z); z <= Math.floor(max.z); z++) {

Block block = worldManager.getBlockIfLoader(x, y, z);

if (block != null && block.isSolid()) {

System.out.println("COLISIÓN DETECTADA en: (" + x + "," + y + "," + z + ")");

return true;

}

}

}

}

return false;

}

private void cleanup() {

worldManager.cleanup();

shaderProgram.cleanup();

textRenderer.cleanup();

grassT.cleanup();

dirtT.cleanup();

stoneT.cleanup();

chunkLoader.terminate();

glfwDestroyWindow(window);

glfwTerminate();

glfwSetErrorCallback(null).free();

}

}

public class WorldManager {

// Mapa para almacenar chunks con clave: "chunkX,chunkY,chunkZ"

final Map<String, Chunk> chunks = new HashMap<>();

private static final int LOAD\_RADIUS = 3;

public Camera camera;

public static File getWorldFolder() {

return worldFolder;

}

private final ShaderProgram shader;

//TEXTURAAAAS

private Texture grass;

private Texture dirt;

private Texture stone;

//Colas de hilos

protected static final ConcurrentLinkedQueue<Chunk> chunksToAdd = new ConcurrentLinkedQueue<>();

protected static final ConcurrentLinkedQueue<Chunk> chunksToMesh = new ConcurrentLinkedQueue<>();

protected static final ConcurrentLinkedQueue<Chunk> chunksToIntegrate = new ConcurrentLinkedQueue<>();

public static final File worldFolder = new File("world");

private ChunkLoaderThread chunkLoader;

private ChunkSaveThread saveThread;

private MeshBuilderThread meshBuilder;

public static WorldManager instance;

public WorldManager(Camera camera, ShaderProgram shader, Texture grass, Texture dirt, Texture stone) {

this.camera = camera;

instance = this;

this.shader = shader;

this.grass = grass;

this.dirt = dirt;

this.stone = stone;

if (!worldFolder.exists()) {

worldFolder.mkdirs();

}

saveThread = new ChunkSaveThread(this);

}

public Camera getCamera() {

return camera;

}

public void setCamera (Camera c){

camera = c;

}

public void setMeshBuilder(MeshBuilderThread meshBuilder) {

this.meshBuilder = meshBuilder;

}

/\*\*

\* Obtiene un chunk por coordenadas de chunk, si no existe lo crea y genera.

\*/

public Chunk getChunk(int chunkX, int chunkY, int chunkZ) {

String key = key(chunkX, chunkY, chunkZ);

Chunk chunk = chunks.get(key);

if (chunk == null) {

chunk = new Chunk(chunkX, chunkY, chunkZ);

chunk.generate(); // Genera el bloque con generación simple o procedural

chunk.buildMesh();

chunks.put(key, chunk);

}

return chunk;

}

public Map<String, Chunk> getChunks() {

return chunks;

}

public void addChunk(Chunk chunk) {

chunks.put(chunk.getKey(), chunk);

}

/\*\*

\* Devuelve el bloque en coordenadas globales (x,y,z en bloques). Retorna 0

\* (aire) si fuera de chunks cargados.

\*/

public Block getBlockIfLoader(int x, int y, int z) {

// 1) Calcular correctamente las coordenadas de chunk en X, Y, Z (antes estaba usando x dos veces)

int chunkX = Math.floorDiv(x, Chunk.SIZEx);

int chunkY = 0;

int chunkZ = Math.floorDiv(z, Chunk.SIZEz);

String key = chunkX + "," + chunkY + "," + chunkZ;

Chunk chunk = chunks.get(key);

if (chunk == null) {

// Si no existe el chunk, devolvemos null (o podrías devolver Block.AIR, según tu preferencia)

System.out.println("El Chunk(" + key + ") no encontrado");

return null;

}

int localX = Math.floorMod(x, Chunk.SIZEx);

int localY = Math.floorMod(y, Chunk.SIZEy);

int localZ = Math.floorMod(z, Chunk.SIZEz);

Block b = chunk.getBlock(localX, localY, localZ);

if (b == null) {

System.out.println("El bloque(" + x + "," + y + "," + z + ") es null");

return AIR; // Aseguramos no devolver null

}

return b;

}

public Chunk getChunkWithoutMesh(String key) {

return chunks.get(key);

}

/\*\*

\* Establece un bloque en coordenadas globales. Reconstruye la malla del

\* chunk modificado.

\*/

public void setBlock(int x, int y, int z, Block blockId) {

int chunkX = Math.floorDiv(x, Chunk.SIZEx);

int chunkY = Math.floorDiv(y, Chunk.SIZEy);

int chunkZ = Math.floorDiv(z, Chunk.SIZEz);

Chunk chunk = getChunk(chunkX, chunkY, chunkZ);

if (chunk == null) {

return;

}

int localX = Math.floorMod(x, Chunk.SIZEx);

int localY = Math.floorMod(y, Chunk.SIZEy);

int localZ = Math.floorMod(z, Chunk.SIZEz);

chunk.setBlock(localX, localY, localZ, blockId);

// Reemplazo dinámico de GRASS por DIRT si se coloca un bloque encima

if (blockId != Block.AIR) {

int belowY = y - 1;

if (belowY >= 0) {

Block blockBelow = getBlockIfLoader(x, belowY, z);

if (blockBelow == GRASS) {

setBlock(x, belowY, z, DIRT);

}

}

}

chunk.setDirty(true);

chunk.buildMesh();

saveThread.requestSave(chunk);

}

public void setBlockIfChunkExists(int x, int y, int z, Block blockId) {

int chunkX = Math.floorDiv(x, Chunk.SIZEx);

int chunkY = Math.floorDiv(y, Chunk.SIZEy);

int chunkZ = Math.floorDiv(z, Chunk.SIZEz);

String key = key(chunkX, chunkY, chunkZ);

if (!chunks.containsKey(key)) {

return; // No generes un chunk nuevo

}

Chunk chunk = chunks.get(key);

int localX = Math.floorMod(x, Chunk.SIZEx);

int localY = Math.floorMod(y, Chunk.SIZEy);

int localZ = Math.floorMod(z, Chunk.SIZEz);

chunk.setBlock(localX, localY, localZ, blockId);

// Reemplazo dinámico de GRASS por DIRT si se coloca un bloque encima

if (blockId != Block.AIR) {

int belowY = y - 1;

int upperY = y + 1;

if (belowY >= 0) {

Block blockBelow = getBlockIfLoader(x, belowY, z);

if (blockBelow == GRASS) {

setBlock(x, belowY, z, DIRT);

}

}

if(upperY >= 0){

Block blockUpper = getBlockIfLoader(x, upperY, z);

if(blockUpper.isSolid() && blockUpper != null && blockId == GRASS){

setBlock(x, y, z, DIRT);

}

}

}

chunk.setDirty(true);

chunk.buildMesh();

saveThread.requestSave(chunk);

}

//si tiene soporte

public boolean hasSupport(int x, int y, int z) {

return getBlockIfLoader(x, y - 1, z) != AIR

|| getBlockIfLoader(x + 1, y, z) != AIR

|| getBlockIfLoader(x - 1, y, z) != AIR

|| getBlockIfLoader(x, y, z + 1) != AIR

|| getBlockIfLoader(x, y, z - 1) != AIR;

}

//Limitar chunks

public void updateChunks(Vector3f playerPos) {

int playerChunkX = Math.floorDiv((int) playerPos.x, Chunk.SIZEx);

int playerChunkZ = Math.floorDiv((int) playerPos.z, Chunk.SIZEz);

Set<String> stillNeeded = new HashSet<>();

List<int[]> toLoad = new ArrayList<>();

for (int dx = -LOAD\_RADIUS; dx <= LOAD\_RADIUS; dx++) {

for (int dz = -LOAD\_RADIUS; dz <= LOAD\_RADIUS; dz++) {

int cx = playerChunkX + dx;

int cz = playerChunkZ + dz;

stillNeeded.add(key(cx, 0, cz));

toLoad.add(new int[]{cx, cz});

}

}

// Ordenar por distancia al jugador

toLoad.sort(Comparator.comparingDouble(coord -> {

float dx = playerChunkX - coord[0];

float dz = playerChunkZ - coord[1];

return dx \* dx + dz \* dz;

}));

// Encolar en orden de cercanía

for (int[] coord : toLoad) {

String k = key(coord[0], 0, coord[1]);

if (!chunks.containsKey(k)) {

chunkLoader.requestLoad(coord[0], coord[1]);

}

}

// Descargar chunks fuera del radio

chunks.keySet().removeIf(k -> {

if (!stillNeeded.contains(k)) {

saveChunk(chunks.get(k));

chunks.get(k).cleanup();

return true;

}

return false;

});

}

public void saveChunk(Chunk chunk) {

saveThread.requestSave(chunk);

}

public void saveChunkImmediate(Chunk chunk) throws IOException {

String key = chunk.getKey(); // por ejemplo: "0,0,0"

File fileTmp = new File(worldFolder, key + ".tmp");

File fileDat = new File(worldFolder, key + ".dat");

// 1) Crear el .tmp y escribir sólo las capas no vacías:

try (DataOutputStream out = new DataOutputStream(new FileOutputStream(fileTmp))) {

// 1.1) Detectar qué capas Y tienen al menos un bloque ≠ AIR

List<Integer> nonEmptyLayers = new ArrayList<>();

for (int y = 0; y < Chunk.SIZEy; y++) {

boolean hasBlock = false;

for (int x = 0; x < Chunk.SIZEx && !hasBlock; x++) {

for (int z = 0; z < Chunk.SIZEz && !hasBlock; z++) {

if (chunk.getBlock(x, y, z) != Block.AIR) {

hasBlock = true;

}

}

}

if (hasBlock) {

nonEmptyLayers.add(y);

}

}

// 1.2) Escribir cuántas capas no vacías vamos a guardar

out.writeInt(nonEmptyLayers.size());

// 1.3) Por cada capa Y, escribir primero el índice Y, y luego todos los bloques de esa capa

for (int y : nonEmptyLayers) {

out.writeInt(y); // índice de la capa

for (int x = 0; x < Chunk.SIZEx; x++) {

for (int z = 0; z < Chunk.SIZEz; z++) {

Block b = chunk.getBlock(x, y, z);

out.writeInt(Block.getBlockId(b));

}

}

}

// El try-with-resources se encarga de cerrar 'out' aquí al salir del bloque

}

// 2) Una vez fuera del try, el flujo ya está cerrado. Ahora podemos renombrar.

// Primero borramos el .dat viejo (si existe)

if (fileDat.exists()) {

if (!fileDat.delete()) {

System.err.println("No se pudo borrar el archivo antiguo: " + fileDat.getPath());

// Si quieres, puedes abortar aquí o programar un retry.

}

}

// 3) Finalmente renombrar .tmp → .dat

if (!fileTmp.renameTo(fileDat)) {

System.err.println("No se pudo renombrar " + fileTmp.getName() + " a " + fileDat.getName());

// Opcional: podrías volver a intentar más tarde, o moverlo a otra carpeta “retry/”

}

}

public boolean loadChunk(Chunk chunk) {

String filename = chunk.getKey() + ".dat";

File file = new File(worldFolder, filename);

File tmp = new File(worldFolder, chunk.getKey() + ".tmp");

if (!file.exists() || tmp.exists()) {

return false;

}

try (DataInputStream in = new DataInputStream(new FileInputStream(file))) {

// Inicializar todo con AIR primero para evitar nulos

for (int x = 0; x < Chunk.SIZEx; x++) {

for (int y = terrainHeight; y < Chunk.SIZEy; y++) {

for (int z = 0; z < Chunk.SIZEz; z++) {

chunk.setBlock(x, y, z, Block.AIR);

}

}

}

int layerCount = in.readInt(); // cuántas capas hay

for (int i = 0; i < layerCount; i++) {

int y = in.readInt(); // índice de la capa

for (int x = 0; x < Chunk.SIZEx; x++) {

for (int z = 0; z < Chunk.SIZEz; z++) {

int id = in.readInt();

Block b = Block.getBlockById(id);

chunk.setBlock(x, y, z, b);

}

}

}

return true;

} catch (IOException e) {

System.err.println(" Error leyendo chunk " + chunk.getKey() + ": " + e.getMessage());

return false;

}

}

public Chunk pollChunkToMesh() {

return chunksToMesh.poll();

}

public void loadOrGenerateChunk(int chunkX, int chunkZ) {

String key = key(chunkX, 0, chunkZ);

// Si ya está cargado en memoria, no hacemos nada.

if (chunks.containsKey(key)) {

return;

}

// Si ya se pidió su carga (o está en proceso), tampoco hacemos nada.

if (chunkLoader.hasPendingRequest(chunkX, chunkZ)) {

return;

}

//Reinicio para evitar errores de caida

// Comprobación del chunk del jugador

int playerChunkX = Math.floorDiv((int) camera.getPosition().x, Chunk.SIZEx);

int playerChunkZ = Math.floorDiv((int) camera.getPosition().z, Chunk.SIZEz);

//if (chunkX == playerChunkX && chunkZ == playerChunkZ) {

// Game.gravityDelayTime = Game.GRAVITY\_COUNTDOWN\_TIME;

//}

// Si no, pedimos al hilo de carga que lo maneje

chunkLoader.requestLoad(chunkX, chunkZ);

}

public void integrateLoadedChunks() {

synchronized (chunksToAdd) {

for (Chunk chunk : chunksToAdd) {

if (chunk == null) {

continue;

}

//Reinicio para evitar errores de caida

// Comprobación del chunk del jugador

int playerChunkX = Math.floorDiv((int) camera.getPosition().x, Chunk.SIZEx);

int playerChunkZ = Math.floorDiv((int) camera.getPosition().z, Chunk.SIZEz);

//if (chunk.chunkX == playerChunkX && chunk.chunkZ == playerChunkZ) {

// Game.gravityDelayTime = Game.GRAVITY\_COUNTDOWN\_TIME;

//}

chunk.buildMesh(); // ahora sí con GL activo

chunks.put(chunk.getKey(), chunk);

}

chunksToAdd.clear();

}

}

public void setChunkLoader(ChunkLoaderThread loader) {

this.chunkLoader = loader;

}

public void enqueueChunkToAdd(Chunk c) {

synchronized (chunksToAdd) {

chunksToAdd.add(c);

}

}

public void enqueueChunkToMesh(Chunk c) {

chunksToMesh.add(c);

}

public void enqueueChunkToIntegrate(Chunk c) {

chunksToIntegrate.add(c);

}

/\*\*

\* Renderiza todos los chunks cargados

\*/

public void render() {

// 1) Calcula proyección y vista (o mantenlas en campos si no cambian cada frame)

Matrix4f projection = camera.getProjectionMatrix(70, 1280f / 720f, 0.01f, 100f);

Matrix4f view = camera.getViewMatrix();

// 2) Bindea TU shader antes de setear uniforms

shader.bind();

// Ahora sí, OpenGL sabe que “viewPos” y “lightPos” van a este shader

//shader.setUniform3f("viewPos", camera.getEyePosition());

shader.setUniform3f("lightPos", new Vector3f(-0.5f, -1.0f, -0.5f).normalize());

shader.setUniformMat4("view", view);

shader.setUniformMat4("projection", projection);

// (También podrías setear aquí las matrices view/projection si no lo haces dentro de Chunk.render)

// 3) Dibuja todos los chunks

for (Chunk chunk : chunks.values()) {

// Chunk.render seguirá bind() y unbind() internamente

// pero como tú ya bindiaste el shader aquí, al menos los uniforms ya existen

chunk.render(projection, view, shader, grass, dirt, stone);

}

// 4) Desbindea el shader cuando termines

shader.unbind();

}

private String key(int x, int y, int z) {

return x + "," + y + "," + z;

}

/\*\*

\* Limpia todos los recursos de chunks

\*/

public void cleanup() {

for (Chunk chunk : chunks.values()) {

chunk.cleanup();

}

chunks.clear();

}

public void startThreads() {

meshBuilder.start();

saveThread.start();

}

}

public class Block {

private BlockType type;

public boolean isSolid;

//Bloques

public static final Block AIR = new Block(false, BlockType.AIR);

public static final Block DIRT = new Block(true, BlockType.DIRT);

public static final Block GRASS = new Block(true, BlockType.GRASS);

public static final Block STONE = new Block(true, BlockType.STONE);

public static final Block[] BLOCKS = new Block[256];

static {

BLOCKS[0] = AIR;

BLOCKS[1] = GRASS;

BLOCKS[2] = DIRT;

BLOCKS[3] = STONE;

}

public Block(boolean isSolid, BlockType type) {

this.isSolid = isSolid;

this.type = type;

}

public BlockType getType() {

return type;

}

public boolean isSolid() {

return isSolid;

}

public static int getBlockId(Block block) {

for (int i = 0; i < BLOCKS.length; i++) {

if (Block.BLOCKS[i] == block) {

return i;

}

}

return 0;

}

public static Block getBlockById(int id) {

if (id >= 0 && id < Block.BLOCKS.length) {

return BLOCKS[id];

}

return AIR;

}

public Texture getTexture() throws Exception {

if (this == GRASS) {

return new Texture("res/textures/grass\_texture.png");

}

if (this == DIRT) {

return new Texture("res/textures/dirt\_texture.png");

}

if (this == STONE) {

return new Texture("res/textures/stone\_texture.png");

} else {

return null;

}

}

}

public class BlockPreviewRenderer {

private final CubeSelected cube;

private final ShaderProgram shader;

public BlockPreviewRenderer(ShaderProgram shader) throws Exception {

this.shader = shader;

this.cube = new CubeSelected(); // la misma clase usada para pruebas

}

public void render(Texture texture, int x, int y, int width, int height, int windowWidth, int windowHeight) {

Matrix4f projection = new Matrix4f().ortho(0, windowWidth, windowHeight, 0, -1, 1);

Matrix4f view = new Matrix4f(); // identidad

Matrix4f model = new Matrix4f().translate(x, y, 0).scale(width, height, 1f);

shader.bind();

shader.setUniformMat4("projection", projection);

shader.setUniformMat4("view", view);

shader.setUniformMat4("model", model);

glActiveTexture(GL\_TEXTURE0);

texture.bind();

shader.setUniform1i("textureSampler", 0);

cube.render();

shader.unbind();

}

public void cleanup() {

cube.cleanup();

}

}

public class BlockSelector {

private Vector3i selectedPos = new Vector3i(0, -999, 0);

private MeshBorder outlineMesh;

public BlockSelector() {

float[] lines = createWireframeCube();

outlineMesh = new MeshBorder(lines);

}

public void setPos(Vector3i Pos) {

this.selectedPos.set(Pos);

}

private float[] createWireframeCube() {

return new float[]{

// Base (Y = 0)

0, 0, 0, 1, 0, 0, // Línea 1: (0,0,0) → (1,0,0)

1, 0, 0, 1, 0, 1, // Línea 2: (1,0,0) → (1,0,1)

1, 0, 1, 0, 0, 1, // Línea 3: (1,0,1) → (0,0,1)

0, 0, 1, 0, 0, 0, // Línea 4: (0,0,1) → (0,0,0)

// Techo (Y = 1)

0, 1, 0, 1, 1, 0, // Línea 5: (0,1,0) → (1,1,0)

1, 1, 0, 1, 1, 1, // Línea 6: (1,1,0) → (1,1,1)

1, 1, 1, 0, 1, 1, // Línea 7: (1,1,1) → (0,1,1)

0, 1, 1, 0, 1, 0, // Línea 8: (0,1,1) → (0,1,0)

// Aristas verticales

0, 0, 0, 0, 1, 0, // Línea 9: (0,0,0) → (0,1,0)

1, 0, 0, 1, 1, 0, // Línea 10: (1,0,0) → (1,1,0)

1, 0, 1, 1, 1, 1, // Línea 11: (1,0,1) → (1,1,1)

0, 0, 1, 0, 1, 1 // Línea 12: (0,0,1) → (0,1,1)

};

}

public void render(Vector3f pos, Matrix4f projection, Matrix4f view, ShaderProgram shader) {

shader.bind();

Matrix4f model = new Matrix4f().translate(pos).scale(1.0f);

Matrix4f mvp = new Matrix4f(projection).mul(view).mul(model);

shader.setUniformMat4("mvp", mvp);

glBindVertexArray(outlineMesh.getVaoId());

glEnableVertexAttribArray(0);

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

glEnable(GL\_POLYGON\_OFFSET\_LINE);

glPolygonOffset(-1.0f, -1.0f);

outlineMesh.render();

glDisable(GL\_POLYGON\_OFFSET\_LINE);

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);

glDisableVertexAttribArray(0);

glBindVertexArray(0);

shader.unbind();

}

public void cloanUp() {

outlineMesh.cleanup();

}

}

public enum BlockType {

AIR(false),

DIRT(true),

GRASS(true),

STONE(true);

private final boolean isSolid;

BlockType(boolean isSolid) {

this.isSolid = isSolid;

}

public boolean isSolid() {

return isSolid;

}

}

public class Camera {

private Vector3f position;

private float pitch; // Rotación vertical

private float yaw; // Rotación horizontal

private final Vector3f front;

private final Vector3f up;

private final Vector3f right;

private final Vector3f worldUp;

public static final float PLAYER\_WIDTH = 0.5f;

public static final float PLAYER\_HEIGHT = 1.8f;

public Camera() {

position = new Vector3f();

pitch = 0.0f;

yaw = -90.0f; // mirando hacia adelante (eje -Z)

front = new Vector3f(0, 0, -1);

up = new Vector3f(0, 1, 0);

right = new Vector3f(1, 0, 0);

worldUp = new Vector3f(0, 1, 0);

updateCameraVectors();

}

public Matrix4f getViewMatrix() {

Vector3f eye = new Vector3f(position).add(0,1.6f,0); //posicion cabeza

return new Matrix4f().lookAt(eye, new Vector3f(eye).add(front),new Vector3f(0,1,0));

}

public Matrix4f getProjectionMatrix(float fov, float aspect, float near, float far) {

return new Matrix4f().perspective((float) Math.toRadians(fov), aspect, near, far);

}

public void processKeyboard(Vector3f direction, float speed) {

Vector3f velocity = new Vector3f(direction).mul(speed);

position.add(velocity);

}

public void processMouseMovement(float xoffset, float yoffset, float sensitivity) {

yaw += xoffset \* sensitivity;

pitch -= yoffset \* sensitivity;

// Limitar la inclinación vertical

if (pitch > 89.0f) {

pitch = 89.0f;

}

if (pitch < -89.0f) {

pitch = -89.0f;

}

updateCameraVectors();

}

private void updateCameraVectors() {

// Calcula la dirección del frente con yaw/pitch

front.x = (float) Math.cos(Math.toRadians(yaw)) \* (float) Math.cos(Math.toRadians(pitch));

front.y = (float) Math.sin(Math.toRadians(pitch));

front.z = (float) Math.sin(Math.toRadians(yaw)) \* (float) Math.cos(Math.toRadians(pitch));

front.normalize();

right.set(front).cross(worldUp).normalize();

up.set(right).cross(front).normalize();

}

public Vector3f getPosition() {

return position;

}

public Vector3f getFront() {

return front;

}

public Vector3f getRight() {

return right;

}

public Vector3f getUp() {

return up;

}

public void moveForward(float speed) {

position.add(new Vector3f(front).mul(speed));

}

public void moveBackward(float speed) {

position.sub(new Vector3f(front).mul(speed));

}

public void moveLeft(float speed) {

position.sub(new Vector3f(right).mul(speed));

}

public void moveRight(float speed) {

position.add(new Vector3f(right).mul(speed));

}

public void moveUp(float speed) {

position.add(new Vector3f(worldUp).mul(speed));

}

public void moveDown(float speed) {

position.sub(new Vector3f(worldUp).mul(speed));

}

public Vector3f getBoundingBoxMin() {

return new Vector3f(

position.x - PLAYER\_WIDTH / 2f,

position.y,

position.z - PLAYER\_WIDTH / 2f

);

}

public Vector3f getBoundingBoxMax() {

return new Vector3f(

position.x + PLAYER\_WIDTH / 2f,

position.y + PLAYER\_HEIGHT,

position.z + PLAYER\_WIDTH / 2f

);

}

public void setPosition(float x, float y, float z){

position = new Vector3f(x, y, z);

}

public Vector3f getEyePosition(){

return new Vector3f(position).add(0,1.6f,0);

}

}

public class Chunk {

// … campos existentes …

private Mesh grassMesh;

private Mesh dirtMesh;

private Mesh stoneMesh;

public static final int SIZEx = 16;

public static final int SIZEy = 16;

public static final int SIZEz = 16;

public static final int terrainHeight = 4;

private boolean dirty = false;

public final int chunkX, chunkY, chunkZ;

private final Block[][][] blocks = new Block[SIZEx][SIZEy][SIZEz];

public Chunk(int chunkX, int chunkY, int chunkZ) {

this.chunkX = chunkX;

this.chunkY = chunkY;

this.chunkZ = chunkZ;

}

/\*\*

\* Generación simple de bloques, por ahora piso sólido hasta y = 4

\*/

public void generate() {

for (int x = 0; x < SIZEx; x++) {

for (int y = 0; y < SIZEy; y++) {

for (int z = 0; z < SIZEz; z++) {

if (y == terrainHeight) {

blocks[x][y][z] = Block.GRASS;

} else if (y < terrainHeight && y > 0) {

blocks[x][y][z] = DIRT;

} else if (y == 0) {

blocks[x][y][z] = STONE;

} else {

blocks[x][y][z] = AIR;

}

}

}

}

}

//Si el cubo esta de cara al cielo

public boolean isExposedToSky(int x, int y, int z) {

for (int ty = y + 1; ty < SIZEy; ty++) {

if (blocks[x][ty][z] != null && blocks[x][ty][z].isSolid()) {

return false; // hay un bloque encima

}

}

return true;

}

/\*\*

\* Establece un bloque dentro del chunk local

\*/

public void setBlock(int x, int y, int z, Block blockId) {

if (blockId == null) {

System.err.println("ERROR! se intentra colocar un bloque null en (" + x + "," + y + "," + z + ")");

return;

}

if (x >= 0 && x < SIZEx && y >= 0 && y < SIZEy && z >= 0 && z < SIZEz) {

blocks[x][y][z] = blockId;

dirty = true;

if(!blockId.getType().equals(AIR) && blockId != null){

System.out.println("Bloque sobrescrito en" + x + "," + y + "," + z + "con " + blockId.getType());

}

}

}

public boolean isDirty() {

return dirty;

}

public void setDirty(boolean flag) {

this.dirty = flag;

}

/\*\*

\* Obtiene el bloque en coordenadas locales del chunk

\*/

public Block getBlock(int x, int y, int z) {

if (x < 0 || x >= SIZEx || y < 0 || y >= SIZEy || z < 0 || z >= SIZEz) {

return AIR;

}

if (blocks[x][y][z] == null) {

return AIR;

}

if (x >= 0 && x < SIZEx && y >= 0 && y < SIZEy && z >= 0 && z < SIZEz) {

return blocks[x][y][z];

}

return AIR;

}

/\*\*

\* Construye o reconstruye la malla del chunk aplicando ocultación de caras

\*/

public void buildMesh() {

// 1) Tres listas de floats para GRASS, DIRT y STONE:

List<Float> grassVerts = new ArrayList<>();

List<Float> dirtVerts = new ArrayList<>();

List<Float> stoneVerts = new ArrayList<>();

for (int x = 0; x < SIZEx; x++) {

for (int y = 0; y < SIZEy; y++) {

for (int z = 0; z < SIZEz; z++) {

Block block = blocks[x][y][z];

if (block != null && block.isSolid()) {

float[] cube = CubeGenerator.createCube(x, y, z, blocks);

// ¿De qué tipo es este bloque?

switch (block.getType()) {

case GRASS:

for (float f : cube) {

grassVerts.add(f);

}

break;

case DIRT:

for (float f : cube) {

dirtVerts.add(f);

}

break;

case STONE:

for (float f : cube) {

stoneVerts.add(f);

}

break;

default:

// (si añades más tipos, agrégalos aquí)

}

}

}

}

}

// 2) Convertir las tres listas a tres arrays float[]

if (!grassVerts.isEmpty()) {

float[] gArr = new float[grassVerts.size()];

for (int i = 0; i < gArr.length; i++) {

gArr[i] = grassVerts.get(i);

}

if (grassMesh != null) {

grassMesh.cleanup();

}

grassMesh = new Mesh(gArr);

} else {

grassMesh = null;

}

if (!dirtVerts.isEmpty()) {

float[] dArr = new float[dirtVerts.size()];

for (int i = 0; i < dArr.length; i++) {

dArr[i] = dirtVerts.get(i);

}

if (dirtMesh != null) {

dirtMesh.cleanup();

}

dirtMesh = new Mesh(dArr);

} else {

dirtMesh = null;

}

if (!stoneVerts.isEmpty()) {

float[] sArr = new float[stoneVerts.size()];

for (int i = 0; i < sArr.length; i++) {

sArr[i] = stoneVerts.get(i);

}

if (stoneMesh != null) {

stoneMesh.cleanup();

}

stoneMesh = new Mesh(sArr);

} else {

stoneMesh = null;

}

// 3) Limpiar la “antigua” malla completa (si la tenías)

// (opcional, en tu caso ya no usas 'this.mesh' sino las tres nuevas)

}

/\*\*

\* Verifica si un bloque vecino es aire o está fuera del chunk

\*/

public boolean isTransparent(int x, int y, int z) {

if (x < 0 || x >= SIZEx || y < 0 || y >= SIZEy || z < 0 || z >= SIZEz) {

return true; // Borde del chunk

}

return blocks[x][y][z] == Block.AIR;

}

public String getKey() {

return chunkX + ",0," + chunkZ;

}

public void render(Matrix4f projection, Matrix4f view, ShaderProgram shader,

Texture grassTex,

Texture dirtTex,

Texture stoneTex) {

// PROYECCIÓN y VISTA ya las pasaste desde WorldManager

Matrix4f model = new Matrix4f().translate(

chunkX \* SIZEx,

chunkY \* SIZEy,

chunkZ \* SIZEz

);

Matrix4f mvp = new Matrix4f(projection).mul(view).mul(model);

// 1) Dibuja GRASS

if (grassMesh != null) {

shader.bind();

shader.setUniformMat4("model", model);

shader.setUniformMat4("mvp", mvp);

glActiveTexture(GL\_TEXTURE0);

grassTex.bind();

shader.setUniform1i("textureSampler", 0);

grassMesh.render();

shader.unbind();

}

// 2) Dibuja DIRT

if (dirtMesh != null) {

shader.bind();

shader.setUniformMat4("model", model);

shader.setUniformMat4("mvp", mvp);

glActiveTexture(GL\_TEXTURE0);

dirtTex.bind();

shader.setUniform1i("textureSampler", 0);

dirtMesh.render();

shader.unbind();

}

// 3) Dibuja STONE

if (stoneMesh != null) {

shader.bind();

shader.setUniformMat4("model", model);

shader.setUniformMat4("mvp", mvp);

glActiveTexture(GL\_TEXTURE0);

stoneTex.bind();

shader.setUniform1i("textureSampler", 0);

stoneMesh.render();

shader.unbind();

}

}

public Mesh getMeshGrass() {

return grassMesh;

}

public Mesh getMeshDirt() {

return dirtMesh;

}

public Mesh getMeshStone() {

return stoneMesh;

}

public Mesh getMeshByBlockType(BlockType type) {

if (type == BlockType.DIRT) {

return dirtMesh;

}

if (type == BlockType.GRASS) {

return grassMesh;

}

if (type == BlockType.STONE) {

return stoneMesh;

}

return null;

}

public void cleanup() {

if (grassMesh != null) {

grassMesh.cleanup();

}

if (dirtMesh != null) {

dirtMesh.cleanup();

}

if (stoneMesh != null) {

stoneMesh.cleanup();

}

}

}

public class ChunkLoaderThread extends Thread {

/\*\*

\* Cola de coordenadas (chunkX, chunkZ) pendientes de cargar/generar

\*/

private final Queue<Vector2i> pending = new ConcurrentLinkedQueue<>();

private final ConcurrentHashMap<Vector2i, Boolean> pendingMap = new ConcurrentHashMap<>();

private int count = 1;

/\*\*

\* Referencia al WorldManager, para delegar carga/generación y encolar el

\* chunk

\*/

private final WorldManager worldManager;

/\*\*

\* Bandera para detener el hilo de forma limpia

\*/

private volatile boolean running = true;

public ChunkLoaderThread(WorldManager worldManager) {

this.worldManager = worldManager;

setName("ChunkLoaderThread");

setDaemon(true);

}

/\*\*

\* Solicita la carga o generación de un chunk (solo lo encola). No vuelve a

\* encolar la misma coordenada si ya está pendiente.

\*/

public void requestLoad(int chunkX, int chunkZ) {

Vector2i key = new Vector2i(chunkX, chunkZ);

if (!pendingMap.containsKey(key) && !isAlreadyLoaded(chunkX, chunkZ)) {

pending.add(key);

pendingMap.put(key, true);

}

}

public boolean isAlreadyLoaded(int ChunkX, int ChunkZ) {

return worldManager.getChunks().containsKey(ChunkX + ",0," + ChunkZ);

}

/\*\*

\* Marca el hilo para que deje de ejecutarse y lo interrumpe si está dormido

\*/

public void terminate() {

running = false;

this.interrupt();

}

public boolean hasPendingRequest(int chunkX, int chunkZ) {

return pending.contains(new Vector2i(chunkX, chunkZ));

}

@Override

public void run() {

while (running) {

int processed = 0;

// Procesamos hasta 3 chunks por iteración (prioridad a los más cercanos)

while (processed < 3) {

Vector2i pos = pending.poll();

if (pos == null) {

break; // No hay más peticiones en esta ronda

}

String key = pos.x + ",0," + pos.y;

if (worldManager.getChunks().containsKey(key)) {

// Ya está cargado, lo saltamos

pendingMap.remove(pos);

continue;

}

// Si no está cargado, lo generamos

Chunk chunk = new Chunk(pos.x, 0, pos.y);

boolean loadedFromDisk = worldManager.loadChunk(chunk);

if (!loadedFromDisk) {

chunk.generate();

}

worldManager.enqueueChunkToMesh(chunk);

pendingMap.remove(pos);

pendingMap.remove(pos);

processed++;

}

// Dormimos un breve rato para no saturar la CPU

try {

Thread.sleep(10);

} catch (InterruptedException ignored) {

// Si nos interrumpen, revisamos la bandera 'running' en la siguiente vuelta

}

}

}

}

public class ChunkSaveThread extends Thread {

private final WorldManager worldManager;

// Cola concurrente de chunks pendientes de guardado

private final Set<Chunk> pendingSaves = ConcurrentHashMap.newKeySet();

private volatile boolean running = true;

public ChunkSaveThread(WorldManager worldManager) {

this.worldManager = worldManager;

setName("ChunkSaveThread");

setDaemon(true); // para que no impida cerrar la app

}

/\*\*

\* Encolar un chunk para guardarlo en disco. Si ya está en la cola, no se

\* vuelve a encolar para evitar duplicados.

\*/

public void requestSave(Chunk chunk) {

pendingSaves.add(chunk);

}

@Override

public void run() {

while (running) {

Chunk chunk = null;

Iterator<Chunk> it = pendingSaves.iterator();

if (it.hasNext()) {

chunk = it.next();

it.remove(); // Importante: eliminar antes de guardar

}

if (chunk != null) {

try {

worldManager.saveChunkImmediate(chunk);

} catch (IOException ex) {

System.err.println("Error al guardar el chunk " + chunk.getKey() + ": " + ex.getMessage());

ex.printStackTrace();

}

} else {

// Si no hay chunks por guardar, duerme un poco

try {

Thread.sleep(50); // Reduce uso de CPU

} catch (InterruptedException ignored) {

}

}

}

}

public void terminate() {

running = false;

this.interrupt();

}

}

public class Cube {

private final Mesh mesh;

private Vector3f rotation = new Vector3f();

private Texture texture;

private float[] vertices;

public Cube() throws Exception {

vertices = new float[]{

// Cara frontal (Columna 1, Fila 0)

-0.5f, 0.5f, 0.5f, 0f, -1f, 0f, 0.33f, 0.00f,1.0f,1.0f,

-0.5f, -0.5f, 0.5f, 0f, -1f, 0f, 0.33f, 0.50f,1.0f,1.0f,

0.5f, -0.5f, 0.5f, 0f, -1f, 0f, 0.66f, 0.50f,1.0f,1.0f,

0.5f, -0.5f, 0.5f, 0f, -1f, 0f, 0.66f, 0.50f,1.0f,1.0f,

0.5f, 0.5f, 0.5f, 0f, -1f, 0f, 0.66f, 0.00f,1.0f,1.0f,

-0.5f, 0.5f, 0.5f, 0f, -1f, 0f, 0.33f, 0.00f,1.0f,1.0f,

// Cara trasera (Columna 1, Fila 1)

0.5f, 0.5f, -0.5f, 0f, 0f, -1f, 0.66f, 0.50f,1.0f,1.0f,

0.5f, -0.5f, -0.5f, 0f, 0f, -1f, 0.66f, 1.00f,1.0f,1.0f,

-0.5f, -0.5f, -0.5f, 0f, 0f, -1f, 0.33f, 1.00f,1.0f,1.0f,

-0.5f, -0.5f, -0.5f, 0f, 0f, -1f, 0.33f, 1.00f,1.0f,1.0f,

-0.5f, 0.5f, -0.5f, 0f, 0f, -1f, 0.33f, 0.50f,1.0f,1.0f,

0.5f, 0.5f, -0.5f, 0f, 0f, -1f, 0.66f, 0.50f,1.0f,1.0f,

// Cara izquierda (Columna 0, Fila 1)

-0.5f, 0.5f, -0.5f, -1f, 0f, 0f, 0.00f, 0.50f,1.0f,1.0f,

-0.5f, -0.5f, -0.5f, -1f, 0f, 0f, 0.00f, 1.00f,1.0f,1.0f,

-0.5f, -0.5f, 0.5f, -1f, 0f, 0f, 0.33f, 1.00f,1.0f,1.0f,

-0.5f, -0.5f, 0.5f, -1f, 0f, 0f, 0.33f, 1.00f,1.0f,1.0f,

-0.5f, 0.5f, 0.5f, -1f, 0f, 0f, 0.33f, 0.50f,1.0f,1.0f,

-0.5f, 0.5f, -0.5f, -1f, 0f, 0f, 0.00f, 0.50f,1.0f,1.0f,

// Cara derecha (Columna 2, Fila 1)

0.5f, 0.5f, 0.5f, 1f, 0f, 0f, 0.66f, 0.50f,1.0f,1.0f,

0.5f, -0.5f, 0.5f, 1f, 0f, 0f, 0.66f, 1.00f,1.0f,1.0f,

0.5f, -0.5f, -0.5f, 1f, 0f, 0f, 1.00f, 1.00f,1.0f,1.0f,

0.5f, -0.5f, -0.5f, 1f, 0f, 0f, 1.00f, 1.00f,1.0f,1.0f,

0.5f, 0.5f, -0.5f, 1f, 0f, 0f, 1.00f, 0.50f,1.0f,1.0f,

0.5f, 0.5f, 0.5f, 1f, 0f, 0f, 0.66f, 0.50f,1.0f,1.0f,

// Cara superior (Columna 0, Fila 0)

-0.5f, 0.5f, -0.5f, 0f, 1f, 0f, 0.00f, 0.00f,1.0f,1.0f,

-0.5f, 0.5f, 0.5f, 0f, 1f, 0f, 0.00f, 0.50f,1.0f,1.0f,

0.5f, 0.5f, 0.5f, 0f, 1f, 0f, 0.33f, 0.50f,1.0f,1.0f,

0.5f, 0.5f, 0.5f, 0f, 1f, 0f, 0.33f, 0.50f,1.0f,1.0f,

0.5f, 0.5f, -0.5f, 0f, 1f, 0f, 0.33f, 0.00f,1.0f,1.0f,

-0.5f, 0.5f, -0.5f, 0f, 1f, 0f, 0.00f, 0.00f,1.0f,1.0f,

// Cara inferior (Columna 2, Fila 0)

-0.5f, -0.5f, 0.5f, 0f, -1f, 0f, 0.66f, 0.50f,1.0f,1.0f,

-0.5f, -0.5f, -0.5f, 0f, -1f, 0f, 0.66f, 0.00f,1.0f,1.0f,

0.5f, -0.5f, -0.5f, 0f, -1f, 0f, 1.00f, 0.00f,1.0f,1.0f,

0.5f, -0.5f, -0.5f, 0f, -1f, 0f, 1.00f, 0.00f,1.0f,1.0f,

0.5f, -0.5f, 0.5f, 0f, -1f, 0f, 1.00f, 0.50f,1.0f,1.0f,

-0.5f, -0.5f, 0.5f, 0f, -1f, 0f, 0.66f, 0.50f,1.0f,1.0f

};

mesh = new Mesh(vertices);

}

public void setRotation(Vector3f rotation) {

this.rotation.set(rotation);

}

public void render(Texture texture, Matrix4f projection, Matrix4f view, ShaderProgram shader) {

this.texture = texture;

shader.bind();

Matrix4f model = new Matrix4f()

.translate(-1.7f, 0.7f, 0f)

.scale(0.1f)

.rotateLocalY(0.35f)

.rotateX((float) Math.toRadians(rotation.x))

.rotateY((float) Math.toRadians(rotation.y))

.rotateZ((float) Math.toRadians(rotation.z));

shader.setUniformMat4("model", model);

shader.setUniformMat4("view", view);

shader.setUniformMat4("projection", projection);

glActiveTexture(GL\_TEXTURE0);

this.texture.bind();

shader.setUniform1i("textureSampler", 0);

mesh.render();

shader.unbind();

}

public float[] getVErt() {

return vertices;

}

public void cleanup() {

mesh.cleanup();

}

}

public class CubeFace {

// Vértices por cara, sentido horario desde esquina inferior izquierda

public static float[][] getFaceVertices(Direction dir) {

switch (dir) {

case UP:

return new float[][] {

{0, 1, 1}, {1, 1, 1}, {1, 1, 0}, {0, 1, 0}

};

case DOWN:

return new float[][] {

{0, 0, 0}, {1, 0, 0}, {1, 0, 1}, {0, 0, 1}

};

case FRONT:

return new float[][] {

{0, 0, 1}, {1, 0, 1}, {1, 1, 1}, {0, 1, 1}

};

case BACK:

return new float[][] {

{1, 0, 0}, {0, 0, 0}, {0, 1, 0}, {1, 1, 0}

};

case LEFT:

return new float[][] {

{0, 0, 0}, {0, 0, 1}, {0, 1, 1}, {0, 1, 0}

};

case RIGHT:

return new float[][] {

{1, 0, 1}, {1, 0, 0}, {1, 1, 0}, {1, 1, 1}

};

}

return new float[0][0];

}

public static float[] getNormal(Direction dir) {

switch (dir) {

case UP: return new float[] {0, 1, 0};

case DOWN: return new float[] {0, -1, 0};

case FRONT: return new float[] {0, 0, 1};

case BACK: return new float[] {0, 0, -1};

case LEFT: return new float[] {-1, 0, 0};

case RIGHT: return new float[] {1, 0, 0};

}

return new float[] {0, 0, 0};

}

}

public class CubeGenerator {

// Tamaño de cada celda de la textura (col: 3 columnas, row: 2 filas)

private static final float UV\_WIDTH = 1.0f / 3.0f;

private static final float UV\_HEIGHT = 1.0f / 2.0f;

public enum Direction {

UP(0, +1, 0),

DOWN(0, -1, 0),

FRONT(0, 0, +1),

BACK(0, 0, -1),

LEFT(-1, 0, 0),

RIGHT(+1, 0, 0);

public final int offsetX, offsetY, offsetZ;

Direction(int dx, int dy, int dz) {

this.offsetX = dx;

this.offsetY = dy;

this.offsetZ = dz;

}

}

public static float[] createCube(int cx, int cy, int cz, Block[][][] blocks) {

List<Float> data = new ArrayList<>(6 \* 10 \* 6);

//para cada cara

for (Direction dir : Direction.values()) {

int texCol = getTexCol(dir);

int texRow = getTexRow(dir);

float[][] positions = CubeFace.getFaceVertices(dir);

float[] normal = CubeFace.getNormal(dir);

float u0 = texCol \* UV\_WIDTH, v0 = texRow \* UV\_HEIGHT;

float u1 = u0 + UV\_WIDTH, v1 = v0 + UV\_HEIGHT;

// UVs: se asignan en sentido horario desde esquina inferior izquierda

float[][] uvs = new float[][]{

{u0, v1},

{u1, v1},

{u1, v0},

{u0, v0}

};

int[][] triIndices = new int[][]{

{0, 1, 2},

{2, 3, 0}

};

for (int[] tri : triIndices) {

for (int vi : tri) {

float px = positions[vi][0] + cx;

float py = positions[vi][1] + cy;

float pz = positions[vi][2] + cz;

int nx = (int) positions[vi][0];

int ny = (int) positions[vi][1];

int nz = (int) positions[vi][2];

float shadow = calcShadow(dir, cy);

//AO

int ix = (int) Math.floor(px);

int iy = (int) Math.floor(py);

int iz = (int) Math.floor(pz);

float ao = calAO(nx, ny, nz, dir, blocks, cx, cy, cz);

//UV

float u = uvs[vi][0];

float v = uvs[vi][1];

data.add(px);

data.add(py);

data.add(pz);

data.add(normal[0]);

data.add(normal[1]);

data.add(normal[2]);

data.add(u);

data.add(v);

data.add(shadow);

data.add(ao);

}

}

}

float[] result = new float[data.size()];

for (int i = 0; i < data.size(); i++) {

result[i] = data.get(i);

}

return result;

}

private static float calcShadow(Direction dir, int cy) {

if (cy == 0) {

return 0.1f;

}

if (cy == 1) {

return 0.2f;

}

if (cy == 2) {

return 0.4f;

}

if (cy == 3) {

return 0.6f;

}

if (cy == 4) {

if (dir.equals(dir.UP)) {

return 1.0f;

}

return 0.8f;

}

return 1.0f;

}

private static float calAO(int x, int y, int z, Direction dir, Block[][][] blocks, int ix, int iy, int iz) {

boolean lado1 = false, lado2 = false, esquina = false;

String vertex = x + "," + y + "," + z;

//System.out.println("Es el vertice: ("+vertex+") de la cara "+ dir.toString() );

switch (dir) {

case FRONT:

switch (vertex) {

case "0,0,1":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz + 1);

esquina = isSolidBlock(blocks, ix - 1, iy - 1, iz + 1);

break;

case "1,0,1":

lado1 = isSolidBlock(blocks, ix + 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz + 1);

esquina = isSolidBlock(blocks, ix + 1, iy - 1, iz + 1);

break;

case "1,1,1":

lado1 = isSolidBlock(blocks, ix + 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz + 1);

esquina = isSolidBlock(blocks, ix + 1, iy + 1, iz + 1);

break;

case "0,1,1":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz + 1);

esquina = isSolidBlock(blocks, ix - 1, iy + 1, iz + 1);

break;

}

break;

case BACK:

switch (vertex) {

case "1,0,0":

lado1 = isSolidBlock(blocks, ix + 1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz - 1);

esquina = isSolidBlock(blocks, ix + 1, iy - 1, iz - 1);

break;

case "0,0,0":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz - 1);

esquina = isSolidBlock(blocks, ix - 1, iy - 1, iz - 1);

break;

case "0,1,0":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz + -1);

esquina = isSolidBlock(blocks, ix - 1, iy + 1, iz - 1);

break;

case "1,1,0":

lado1 = isSolidBlock(blocks, ix + 1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz - 1);

esquina = isSolidBlock(blocks, ix + 1, iy + 1, iz - 1);

break;

}

break;

case LEFT:

switch (vertex) {

case "0,0,0":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix - 1, iy - 1, iz);

esquina = isSolidBlock(blocks, ix - 1, iy - 1, iz - 1);

break;

case "0,0,1":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix - 1, iy - 1, iz);

esquina = isSolidBlock(blocks, ix - 1, iy - 1, iz + 1);

break;

case "0,1,1":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix - 1, iy + 1, iz);

esquina = isSolidBlock(blocks, ix - 1, iy + 1, iz + 1);

break;

case "0,1,0":

lado1 = isSolidBlock(blocks, ix - 1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix - 1, iy + 1, iz);

esquina = isSolidBlock(blocks, ix - 1, iy + 1, iz - 1);

break;

}

break;

case RIGHT:

switch (vertex) {

case "1,0,1":

lado1 = isSolidBlock(blocks, ix + 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix + 1, iy - 1, iz);

esquina = isSolidBlock(blocks, ix + 1, iy - 1, iz + 1);

break;

case "1,0,0":

lado1 = isSolidBlock(blocks, ix + 1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix + 1, iy - 1, iz);

esquina = isSolidBlock(blocks, ix + 1, iy - 1, iz - 1);

break;

case "1,1,0":

lado1 = isSolidBlock(blocks, ix+1, iy, iz - 1);

lado2 = isSolidBlock(blocks, ix+1, iy + 1, iz);

esquina = isSolidBlock(blocks, ix+1, iy + 1, iz - 1);

break;

case "1,1,1":

lado1 = isSolidBlock(blocks, ix + 1, iy, iz + 1);

lado2 = isSolidBlock(blocks, ix + 1, iy + 1, iz);

esquina = isSolidBlock(blocks, ix + 1, iy + 1, iz +1);

break;

}

break;

case UP:

switch (vertex) {

case "0,1,1":

lado1 = isSolidBlock(blocks, ix - 1, iy + 1, iz);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz + 1);

esquina = isSolidBlock(blocks, ix - 1, iy + 1, iz + 1);

break;

case "1,1,1":

lado1 = isSolidBlock(blocks, ix + 1, iy + 1, iz);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz + 1);

esquina = isSolidBlock(blocks, ix + 1, iy + 1, iz + 1);

break;

case "1,1,0":

lado1 = isSolidBlock(blocks, ix + 1, iy + 1, iz);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz - 1);

esquina = isSolidBlock(blocks, ix + 1, iy + 1, iz - 1);

break;

case "0,1,0":

lado1 = isSolidBlock(blocks, ix - 1, iy + 1, iz);

lado2 = isSolidBlock(blocks, ix, iy + 1, iz - 1);

esquina = isSolidBlock(blocks, ix - 1, iy + 1, iz - 1);

break;

}

break;

case DOWN:

switch (vertex) {

case "0,0,0":

lado1 = isSolidBlock(blocks, ix - 1, iy - 1, iz);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz - 1);

esquina = isSolidBlock(blocks, ix - 1, iy - 1, iz - 1);

break;

case "1,0,0":

lado1 = isSolidBlock(blocks, ix + 1, iy - 1, iz);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz - 1);

esquina = isSolidBlock(blocks, ix + 1, iy - 1, iz - 1);

break;

case "1,0,1":

lado1 = isSolidBlock(blocks, ix + 1, iy - 1, iz);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz + 1);

esquina = isSolidBlock(blocks, ix + 1, iy - 1, iz + 1);

break;

case "0,0,1":

lado1 = isSolidBlock(blocks, ix - 1, iy - 1, iz);

lado2 = isSolidBlock(blocks, ix, iy - 1, iz + 1);

esquina = isSolidBlock(blocks, ix - 1, iy - 1, iz + 1);

break;

}

break;

}

//ambos lados = sombra total

if (lado1 && lado2) {

return 0.2f;

}

//contamos cuantos de los 3 estan solidos

int totalSolid = 0;

if (lado1) {

totalSolid++;

}

if (lado2) {

totalSolid++;

}

if (esquina) {

totalSolid++;

}

return (3 - totalSolid) / 3.0f;

}

private static boolean isAir(Block[][][] blocks, int x, int y, int z) {

if (x < 0 || x >= blocks.length

|| y < 0 || y >= blocks[0].length

|| z < 0 || z >= blocks[0][0].length) {

return true; // Consideramos aire fuera de los límites

}

if (blocks[x][y][z].isSolid && blocks[x][y][z] != null) {

return false;

}

return true;

}

private static boolean isInShadow(int x, int y, int z, Direction dir) {

final int MAX\_DIST = 8;

for (int i = 1; i <= MAX\_DIST; i++) {

int nx = x + dir.offsetX \* i;

int ny = y + dir.offsetY \* i;

int nz = z + dir.offsetZ \* i;

Block b = WorldManager.instance.getBlockIfLoader(nx, ny, nz);

if (b == null) {

continue;

}

if (b.isSolid()) {

return true;

}

}

return false;

}

/\*\*

\* Comprueba si en las coordenadas locales (x,y,z) existe un bloque sólido.

\* Debemos verificar límites del chunk:

\*/

private static boolean isSolidBlock(Block[][][] blocks, int lx, int ly, int lz) {

if (lx < 0 || lx >= blocks.length

|| ly < 0 || ly >= blocks[0].length

|| lz < 0 || lz >= blocks[0][0].length) {

return false; // fuera de este chunk → considerar “vacío” para AO

}

Block b = blocks[lx][ly][lz];

return (b != null && b.isSolid());

}

private static int getTexCol(Direction dir) {

switch (dir) {

case UP:

return 0;

case DOWN:

return 2;

case FRONT:

return 1;

case BACK:

return 1;

case LEFT:

return 0;

case RIGHT:

return 2;

}

return 0;

}

private static int getTexRow(Direction dir) {

switch (dir) {

case UP:

return 0;

case DOWN:

return 0;

case FRONT:

return 0;

case BACK:

return 1;

case LEFT:

return 1;

case RIGHT:

return 1;

}

return 0;

}

}

public class CubeSelected {

private int vaoId;

private int vboId;

private int vertexCount;

public CubeSelected() {

float[] vertices = {

// Posiciones // Normales // UVs

// Cara frontal

0, 0, 1, 0, 0, 1, 0, 1,

1, 0, 1, 0, 0, 1, 1, 1,

1, 1, 1, 0, 0, 1, 1, 0,

1, 1, 1, 0, 0, 1, 1, 0,

0, 1, 1, 0, 0, 1, 0, 0,

0, 0, 1, 0, 0, 1, 0, 1,

// Cara trasera

1, 0, 0, 0, 0, -1, 0, 1,

0, 0, 0, 0, 0, -1, 1, 1,

0, 1, 0, 0, 0, -1, 1, 0,

0, 1, 0, 0, 0, -1, 1, 0,

1, 1, 0, 0, 0, -1, 0, 0,

1, 0, 0, 0, 0, -1, 0, 1,

// Cara izquierda

0, 0, 0, -1, 0, 0, 0, 1,

0, 0, 1, -1, 0, 0, 1, 1,

0, 1, 1, -1, 0, 0, 1, 0,

0, 1, 1, -1, 0, 0, 1, 0,

0, 1, 0, -1, 0, 0, 0, 0,

0, 0, 0, -1, 0, 0, 0, 1,

// Cara derecha

1, 0, 1, 1, 0, 0, 0, 1,

1, 0, 0, 1, 0, 0, 1, 1,

1, 1, 0, 1, 0, 0, 1, 0,

1, 1, 0, 1, 0, 0, 1, 0,

1, 1, 1, 1, 0, 0, 0, 0,

1, 0, 1, 1, 0, 0, 0, 1,

// Cara superior

0, 1, 1, 0, 1, 0, 0, 1,

1, 1, 1, 0, 1, 0, 1, 1,

1, 1, 0, 0, 1, 0, 1, 0,

1, 1, 0, 0, 1, 0, 1, 0,

0, 1, 0, 0, 1, 0, 0, 0,

0, 1, 1, 0, 1, 0, 0, 1,

// Cara inferior

0, 0, 0, 0, -1, 0, 0, 1,

1, 0, 0, 0, -1, 0, 1, 1,

1, 0, 1, 0, -1, 0, 1, 0,

1, 0, 1, 0, -1, 0, 1, 0,

0, 0, 1, 0, -1, 0, 0, 0,

0, 0, 0, 0, -1, 0, 0, 1

};

vertexCount = vertices.length / 8;

vaoId = glGenVertexArrays();

glBindVertexArray(vaoId);

vboId = glGenBuffers();

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

glBufferData(GL\_ARRAY\_BUFFER, vertices, GL\_STATIC\_DRAW);

// Posición (vec3)

glVertexAttribPointer(0, 3, GL\_FLOAT, false, 8 \* Float.BYTES, 0);

glEnableVertexAttribArray(0);

// Normal (vec3)

glVertexAttribPointer(1, 3, GL\_FLOAT, false, 8 \* Float.BYTES, 3 \* Float.BYTES);

glEnableVertexAttribArray(1);

// UV (vec2)

glVertexAttribPointer(2, 2, GL\_FLOAT, false, 8 \* Float.BYTES, 6 \* Float.BYTES);

glEnableVertexAttribArray(2);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

}

public void render() {

glBindVertexArray(vaoId);

glDrawArrays(GL\_TRIANGLES, 0, vertexCount);

glBindVertexArray(0);

}

public void cleanup() {

glDeleteBuffers(vboId);

glDeleteVertexArrays(vaoId);

}

}

public class FloatArray {

private float[] data;

private int size;

public FloatArray() {

data = new float[1024];

size = 0;

}

public void add(float[] values) {

ensureCapacity(size + values.length);

for (float v : values) {

data[size++] = v;

}

}

public float[] toArray() {

return Arrays.copyOf(data, size);

}

private void ensureCapacity(int minCapacity) {

if (minCapacity > data.length) {

int newCapacity = Math.max(minCapacity, data.length \* 2);

data = Arrays.copyOf(data, newCapacity);

}

}

}

public class MeshBorder {

private final int vaoId;

private final int vboId;

private final int vertexCount;

public MeshBorder(float[] positions) {

vertexCount = positions.length / 3;

vaoId = glGenVertexArrays();

glBindVertexArray(vaoId);

vboId = glGenBuffers();

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

FloatBuffer buffer = BufferUtils.createFloatBuffer(positions.length);

buffer.put(positions).flip();

glBufferData(GL\_ARRAY\_BUFFER, buffer, GL\_STATIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, false, 3 \* Float.BYTES, 0);

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

}

public int getVaoId() {

return vaoId;

}

public int getVboId() {

return vboId;

}

public void render() {

glBindVertexArray(vaoId);

glEnableVertexAttribArray(0);

glDrawArrays(GL\_LINES, 0, vertexCount);

glDisableVertexAttribArray(0);

glBindVertexArray(0);

}

public void cleanup() {

glDisableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glDeleteBuffers(vboId);

glBindVertexArray(0);

glDeleteVertexArrays(vaoId);

}

}

public class MeshBuilderThread extends Thread {

private final WorldManager worldManager;

private volatile boolean running = true;

public MeshBuilderThread(WorldManager wm) {

this.worldManager = wm;

setName("MeshBuilderThread");

setDaemon(true);

}

public void terminate() {

running = false;

this.interrupt();

}

@Override

public void run() {

while (running) {

Chunk chunk = worldManager.chunksToMesh.poll();

if (chunk != null) {

worldManager.enqueueChunkToAdd(chunk);

}else{

try{Thread.sleep(10);}catch(Exception e){}

}

}

}

}

public class RaycastResult {

public final Vector3f blockPos;

public final Vector3f faceNormal;

public RaycastResult(Vector3f blockPos, Vector3f faceNormal) {

this.blockPos = blockPos;

this.faceNormal = faceNormal;

}

}

public class RaycastUtils {

public static RaycastResult raycast(WorldManager world, Vector3f origin, Vector3f direction, float maxDistance) {

Vector3f ray = new Vector3f(direction).normalize();

Vector3f pos = new Vector3f(origin);

int lastX = (int) Math.floor(pos.x);

int lastY = (int) Math.floor(pos.y);

int lastZ = (int) Math.floor(pos.z);

for (int i = 0; i < maxDistance \* 10; i++) {

pos.add(new Vector3f(ray).mul(0.1f));

int x = (int) Math.floor(pos.x);

int y = (int) Math.floor(pos.y);

int z = (int) Math.floor(pos.z);

if ((x != lastX || y != lastY || z != lastZ)

&& world.getBlockIfLoader(x, y, z) != null

&& world.getBlockIfLoader(x, y, z) != Block.AIR) {

Vector3f blockPos = new Vector3f(x, y, z);

Vector3f faceNormal = new Vector3f(lastX - x, lastY - y, lastZ - z);

return new RaycastResult(blockPos, faceNormal);

}

lastX = x;

lastY = y;

lastZ = z;

}

return null;

}

}

public class ShaderProgram {

private final int programId;

public ShaderProgram(String vertexPath, String fragmentPath) {

programId = glCreateProgram();

int vertexShader = createShader(vertexPath, GL\_VERTEX\_SHADER);

int fragmentShader = createShader(fragmentPath, GL\_FRAGMENT\_SHADER);

glAttachShader(programId, vertexShader);

glAttachShader(programId, fragmentShader);

glLinkProgram(programId);

if (glGetProgrami(programId, GL\_LINK\_STATUS) == GL\_FALSE) {

throw new RuntimeException("Error al vincular programa: " + glGetProgramInfoLog(programId));

}

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

}

private int createShader(String path, int type) {

String code = "";

try {

code = new String(Files.readAllBytes(Paths.get(path)));

} catch (IOException e) {

e.printStackTrace();

}

int shaderId = glCreateShader(type);

glShaderSource(shaderId, code);

glCompileShader(shaderId);

if (glGetShaderi(shaderId, GL\_COMPILE\_STATUS) == GL\_FALSE) {

throw new RuntimeException("Error en shader " + path + ": " + glGetShaderInfoLog(shaderId));

}

return shaderId;

}

public void use() {

glUseProgram(programId);

}

public void setUniformMat4(String name, Matrix4f matrix) {

int location = glGetUniformLocation(programId, name);

try (MemoryStack stack = MemoryStack.stackPush()) {

FloatBuffer fb = stack.mallocFloat(16);

matrix.get(fb);

glUniformMatrix4fv(location, false, fb);

}

}

public void setUniform3f(String name, Vector3f value) {

int location = glGetUniformLocation(programId, name);

if (location != -1) {

glUniform3f(location, value.x, value.y, value.z);

} else {

System.err.println("Uniform not found: " + name);

}

}

public void setUniform1i(String name, int value) {

int location = glGetUniformLocation(programId, name);

if (location == -1) {

System.err.println("Warning: uniform '" + name + "' doesn't exist!");

return;

}

glUniform1i(location, value);

}

public void bind() {

glUseProgram(programId);

}

public void unbind() {

glUseProgram(0);

}

public void cleanup() {

glDeleteProgram(programId);

}

}

public class TextRenderer {

private int textureId;

private int width;

private int height;

private Font font;

private Color color;

private String currentText;

public TextRenderer(String text, Font font, Color color) {

this.font = font;

this.color = color;

updateText(text);

}

public void updateText(String newText) {

if (newText.equals(currentText)) {

return; // Evita regenerar si no cambia

}

if (textureId != 0) {

GL11.glDeleteTextures(textureId);

}

BufferedImage img = createTextImage(newText, font, color);

this.width = img.getWidth();

this.height = img.getHeight();

this.textureId = loadTextureFromBufferedImage(img);

this.currentText = newText;

}

private BufferedImage createTextImage(String text, Font font, Color color) {

BufferedImage img = new BufferedImage(256, 64, BufferedImage.TYPE\_INT\_ARGB);

Graphics2D g = img.createGraphics();

g.setComposite(AlphaComposite.Src);

g.setRenderingHint(RenderingHints.KEY\_TEXT\_ANTIALIASING, RenderingHints.VALUE\_TEXT\_ANTIALIAS\_ON);

g.setFont(font);

g.setColor(new Color(255, 255, 255, 255)); // blanco opaco

g.setBackground(new Color(0, 0, 0, 0)); // fondo completamente transparente

g.clearRect(0, 0, img.getWidth(), img.getHeight());

g.drawString(text, 10, 40);

g.dispose();

return img;

}

public static int loadTextureFromBufferedImage(BufferedImage image) {

int[] pixels = new int[image.getWidth() \* image.getHeight()];

image.getRGB(0, 0, image.getWidth(), image.getHeight(), pixels, 0, image.getWidth());

ByteBuffer buffer = BufferUtils.createByteBuffer(image.getWidth() \* image.getHeight() \* 4);

for (int y = 0; y < image.getHeight(); y++) {

for (int x = 0; x < image.getWidth(); x++) {

int pixel = pixels[y \* image.getWidth() + x];

buffer.put((byte) ((pixel >> 16) & 0xFF)); // Red

buffer.put((byte) ((pixel >> 8) & 0xFF)); // Green

buffer.put((byte) (pixel & 0xFF)); // Blue

buffer.put((byte) ((pixel >> 24) & 0xFF)); // Alpha

}

}

buffer.flip();

int textureID = glGenTextures();

glBindTexture(GL\_TEXTURE\_2D, textureID);

// Preservar el canal alfa

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, image.getWidth(), image.getHeight(), 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, buffer);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glBindTexture(GL\_TEXTURE\_2D, 0);

return textureID;

}

public void render(float x, float y) {

glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, textureId);

glColor4f(1, 1, 1, 1);

glBegin(GL\_QUADS);

glTexCoord2f(0, 0);

glVertex2f(x, y);

glTexCoord2f(1, 0);

glVertex2f(x + width, y);

glTexCoord2f(1, 1);

glVertex2f(x + width, y + height);

glTexCoord2f(0, 1);

glVertex2f(x, y + height);

glEnd();

glBindTexture(GL\_TEXTURE\_2D, 0);

glDisable(GL\_TEXTURE\_2D);

}

public void cleanup() {

if (textureId != 0) {

glDeleteTextures(textureId);

}

}

}

public class Texture {

private final int id;

public Texture(String fileName) throws Exception {

id = glGenTextures();

glBindTexture(GL\_TEXTURE\_2D, id);

// Parámetros de textura (filtrado y wrap)

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

// Cargar imagen

try (MemoryStack stack = MemoryStack.stackPush()) {

IntBuffer width = stack.mallocInt(1);

IntBuffer height = stack.mallocInt(1);

IntBuffer channels = stack.mallocInt(1);

// La ruta debe ser relativa al directorio del proyecto o ruta absoluta

ByteBuffer image = STBImage.stbi\_load(fileName, width, height, channels, 4);

if (image == null) {

throw new Exception("Failed to load a texture file!" + System.lineSeparator() + STBImage.stbi\_failure\_reason());

}

int texWidth = width.get(0);

int texHeight = height.get(0);

System.out.println("Texture loaded: " + texWidth + "x" + texHeight);

// Subir textura a OpenGL

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA, width.get(0), height.get(0), 0,

GL\_RGBA, GL\_UNSIGNED\_BYTE, image);

STBImage.stbi\_image\_free(image);

}

}

public int getId() {

return id;

}

public void bind() {

glBindTexture(GL\_TEXTURE\_2D, id);

}

public void cleanup() {

glDeleteTextures(id);

}

}

public class Window {

private final int width;

private final int height;

private final String title;

private long window;

public Window(int width, int height, String title) {

this.width = width;

this.height = height;

this.title = title;

}

public void init() {

// Setup error callback

GLFWErrorCallback.createPrint(System.err).set();

// Init GLFW

if (!glfwInit()) {

throw new IllegalStateException("Unable to initialize GLFW");

}

// Configure GLFW

glfwDefaultWindowHints();

glfwWindowHint(GLFW\_VISIBLE, GLFW\_FALSE); // keep window hidden until after creation

glfwWindowHint(GLFW\_RESIZABLE, GLFW\_TRUE);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GLFW\_TRUE); // for macOS

// Create the window

window = glfwCreateWindow(width, height, title, NULL, NULL);

if (window == NULL) {

throw new RuntimeException("Failed to create the GLFW window");

}

// Center the window

GLFWVidMode vidmode = glfwGetVideoMode(glfwGetPrimaryMonitor());

if (vidmode != null) {

glfwSetWindowPos(

window,

(vidmode.width() - width) / 2,

(vidmode.height() - height) / 2

);

}

// Make the OpenGL context current

glfwMakeContextCurrent(window);

// Enable v-sync

glfwSwapInterval(1);

// Show the window

glfwShowWindow(window);

// Create OpenGL capabilities (must be done after context is current)

GL.createCapabilities();

// Configure OpenGL defaults

glEnable(GL\_DEPTH\_TEST); // important for 3D

glClearColor(0.1f, 0.2f, 0.3f, 1.0f); // nice blue background

glViewport(0, 0, width, height); // set viewport size

}

public void update() {

glfwSwapBuffers(window); // swap the color buffers

glfwPollEvents(); // poll for window events

}

public boolean shouldClose() {

return glfwWindowShouldClose(window);

}

public void cleanup() {

glfwDestroyWindow(window);

glfwTerminate();

glfwSetErrorCallback(null).free();

}

}

//Fragment.glsl

#version 330 core

in vec2 fragTexCoord;

in vec3 fragPos;

in vec3 normal;

in float shadeFactor;

in float vAO;

out vec4 fragColor;

uniform sampler2D textureSampler;

// Dirección de la luz (normalizada), ej: vec3(-0.5, -1.0, -0.5)

uniform vec3 lightPos;

void main() {

vec3 norm = normalize(normal);

// Dirección opuesta a la luz (porque viene "desde el sol")

float diff = max(dot(norm, -lightPos), 0.0);

diff\*=shadeFactor;

vec3 ambient = vec3(0.3)\* diff; // Luz ambiental constante

vec3 diffuse = vec3(0.7);

vec3 texColor = texture(textureSampler, fragTexCoord).rgb;

vec3 shadeDiffuse = diffuse\*shadeFactor;

vec3 lighting = (ambient + shadeDiffuse)\*texColor\*vAO;

fragColor = vec4(lighting, 1.0);

}

//selector\_fragment.glsl

#version 330 core

out vec4 fragColor;

void main() {

fragColor = vec4(1.0, 1.0, 1.0, 1.0); // Blanco puro

}

//selector\_vertex.glsl

#version 330 core

layout(location = 0) in vec3 inPosition;

uniform mat4 mvp;

void main() {

gl\_Position = mvp \* vec4(inPosition, 1.0);

}

//text\_fragment.glsl

#version 330 core

in vec2 TexCoords;

out vec4 FragColor;

uniform sampler2D text; // atlas de caracteres (un solo canal R)

uniform vec3 textColor; // color del texto (RGB)

void main() {

// La textura 'text' tiene sólo el canal R con la información de la glyph

float alpha = texture(text, TexCoords).r;

// Usamos textColor para RGB, y alpha para transparencia

FragColor = vec4(textColor, alpha);

}

//text\_vertex.glsl

#version 330 core

layout (location = 0) in vec4 vertex; // <vec2 pos, vec2 tex>

out vec2 TexCoords;

uniform mat4 projection;

void main() {

// vertex.xy = posición en pantalla (ya en coordenadas ortográficas)

// vertex.zw = coordenadas de textura

gl\_Position = projection \* vec4(vertex.xy, 0.0, 1.0);

TexCoords = vertex.zw;

}

//shader.glsl

#version 330 core

layout(location = 0) in vec3 inPosition;

layout(location = 1) in vec3 inNormal;

layout(location = 2) in vec2 inTexCoord;

layout(location = 3) in float inShade;

layout(location = 4) in float inAO;

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

out vec2 fragTexCoord;

out vec3 fragPos;

out vec3 normal;

out float shadeFactor;

out float vAO;

void main() {

fragTexCoord = inTexCoord;

vec4 worldPos = model \* vec4(inPosition, 1.0);

fragPos = worldPos.xyz;

normal = mat3(transpose(inverse(model))) \* inNormal;

shadeFactor = inShade;

vAO = inAO;

gl\_Position = projection \* view \* worldPos;

}