INFORME TÉCNICO ? FURNACEKNIGHTJAVA (SERVIDOR)

1. Resumen ejecutivo

El repositorio implementa la versión servidor de FurnaceKnightJAVA, un juego 2D construido sobre libGDX. La lógica central vive en el módulo ‘core‘, mientras que ‘lwjgl3‘ contiene el lanzador de escritorio. El servidor actœa como autoridad: simula el mundo, ejecuta la IA, resuelve colisiones y transmite el estado a los clientes mediante UDP. Las decisiones clave del diseæo son:

? Pantallas de libGDX para separar el lanzador, la carga de recursos y el gameplay (‘ServerLauncherScreen‘, ‘LoadingScreen‘, ‘GameScreen‘, ‘InterlevelLoadingScreen‘).
? Una arquitectura de entidades con mÆquinas de estados (‘EntityStateMachine‘) que controla jugadores y enemigos.
? Sincronización ligera basada en mensajes de texto (‘UPDATE_PLAYER‘, ‘CREATE_ENTITY‘, ‘HUD_FIRE_STATE‘, etc.) enviada desde ‘ServerThread‘.
? Un flujo de progresión con mapas Tiled, portales entre salas y un sistema de mejoras financiado con monedas del juego.

2. Arquitectura del proyecto

? **MainGame (‘core/src/main/java/com/FK/game/core/MainGame.java‘)** orquesta el ciclo de vida de libGDX: carga UI skins, crea dos ‘PlayerData‘, inicia la pantalla de lanzamiento y mantiene referencia al ‘ServerThread‘. Un ‘UncaughtExceptionHandler‘ garantiza que cualquier error detendrÆ el servidor limpiamente.
? **Módulo ‘core‘** contiene gameplay, red, UI y assets. **‘lwjgl3‘** expone la aplicación de escritorio que arranca ‘MainGame‘.
? **GameContext (‘core/src/main/java/com/FK/game/core/GameContext.java‘)** centraliza referencias globales a la pantalla activa, la sala del lanzador y la lista de jugadores vivos, ademÆs de asignar IDs de red para nuevas entidades.

3. Flujo de arranque y pantallas

1. **ServerLauncherScreen** ofrece un botón para iniciar el servidor (‘core/src/main/java/com/FK/game/screens/ServerLauncherScreen.java‘). Al pulsarlo crea ‘ServerThread‘, guarda la instancia en ‘MainGame‘ y avanza a ‘LoadingScreen‘.
2. **LoadingScreen (‘core/src/main/java/com/FK/game/screens/LoadingScreen.java‘)** bloquea hasta que ‘Assets‘, ‘AnimationCache‘ y ‘SoundCache‘ cargan todas las texturas y sonidos.
3. **GameScreen (‘core/src/main/java/com/FK/game/screens/GameScreen.java‘)** es la pantalla principal. Configura cÆmara, viewport y HUD, carga mapas Tiled mediante ‘MapManager‘ y genera entidades segœn capas de spawn. Al detectar que se completóuna sala crea un portal que, al tocarlo, manda a los clientes a ‘InterlevelLoadingScreen‘ y selecciona el siguiente mapa.
4. **InterlevelLoadingScreen** (misma ruta) cubre la carga entre salas, elige aleatoriamente el próximo ‘.tmx‘ (con probabilidad especial para salas de jefe tras dos habitaciones) y notifica ‘LEVEL_READY:<mapa>‘ antes de devolver el control a ‘GameScreen‘.

4. Gestión de datos compartidos del jugador

? **PlayerData (‘core/src/main/java/com/FK/game/core/PlayerData.java‘)** guarda monedas, niveles de salud y daæo, y puntos de vida actuales. Sus mØtodos calculan daæo/vida mÆxima y reinician progresión al morir o reiniciar sala.
? **UpgradeManager (‘core/src/main/java/com/FK/game/core/UpgradeManager.java‘)** aplica la economía: calcula costos crecientes y actualiza ‘PlayerData‘. Se usa tanto en el HUD local como en las solicitudes remotas de ‘ServerThread‘.
? **UpgradeWindow y ServerDisconnectWindow (‘core/src/main/java/com/FK/game/ui/*.java‘)** son ventanas Scene2D para gestionar upgrades en la hoguera y finalizar sesiones.

5. Recursos y presentación audiovisual

? **Assets (‘core/src/main/java/com/FK/game/animations/Assets.java‘)** carga texturas bÆsicas y expone referencias estÆticas (jugador idle, monedas, píxel blanco para Rayos del jefe, etc.).
? **AnimationCache (‘core/src/main/java/com/FK/game/core/AnimationCache.java‘)** se encarga de cargar todos los ‘AnimationType‘ (jugador, enemigos, objetos) y entregar ‘AnimationHandler‘ instanciados.
? **UIAssets (‘core/src/main/java/com/FK/game/animations/UIAssets.java‘)** carga el skin ‘glassy-ui‘.
? **SoundCache (‘core/src/main/java/com/FK/game/sounds/SoundCache.java‘)** encapsula los sonidos (‘SoundType‘) y su reproducción espacial/loop, lo que permite disparos direccionales y pasos continuos.

6. Entrada, jugador y HUD

? **InputHandler** abstrae la fuente de entrada (‘KeyboardInputHandler‘ para pruebas locales y ‘NetworkInputHandler‘ para inputs remotos enviados desde ‘ServerThread‘).
? **Player (‘core/src/main/java/com/FK/game/entities/Player.java‘)** extiende ‘CharacterEntity‘ y monta su mÆquina de estados (‘IdleState‘, ‘WalkingState‘, ‘ChargingJumpState‘, etc.). Gestiona cooldown del ataque de fuego, aplica ‘PlayerData‘, mantiene un ‘FireAttackHUD‘ y sincroniza ‘StateMessage‘+‘FacingDirection‘ para la red.
? **FireAttackHUD y CoinHUD (‘core/src/main/java/com/FK/game/screens/*.java‘)** representan el estado de carga del ataque de fuego y el contador de monedas. El HUD usa animaciones dedicadas para mostrar transiciones (carga/descarga) y recibe actualizaciones periódicas enviadas por ‘GameScreen‘.

7. Sistema de entidades y estados

? **Entity (‘core/src/main/java/com/FK/game/entities/Entity.java‘)** define bounding boxes, físicas bÆsicas, cajas de daæo y metadatos de red (‘EntityTypeMessage‘, ‘StateMessage‘, ‘networkId‘).
? **CharacterEntity (‘core/src/main/java/com/FK/game/entities/CharacterEntity.java‘)** aæade salud, knockback e integración con ‘EntityStateMachine‘. Cambia a ‘DamageState‘ o ‘DeathState‘ dependiendo del resultado del ataque.

? **Enemy (‘core/src/main/java/com/FK/game/entities/Enemy.java‘)** carga animaciones específicas (‘EnemyAnimationType‘), detecta jugadores cercanos y maneja ‘attackCooldown‘. Variantes como ‘Bolb‘, ‘Slop‘, ‘Fungop‘ o ‘Boss‘ definen sus propios estados (por ejemplo ‘BolbWalkState‘, ‘BossLaserAttackState‘).
? **State framework (‘core/src/main/java/com/FK/game/states/*.java‘)**: ‘EntityState‘ dicta la interfaz, ‘EntityStateMachine‘ ejecuta ‘handleInput‘+‘update‘ cada frame y ‘StateUtils‘ ayuda con lógica comœn (detección de caída, chequeo de suelo). Las clases de estado cambian animaciones, reproducen sonidos y modifican velocidad.
? **Objetos especiales**: ‘Portal‘ y ‘Fire‘ tienen sus propias mÆquinas de estados (‘PortalState‘, ‘FireBasicState‘) basadas en ‘ObjectsAnimationType‘, mientras que ‘Coin‘ se desplaza hacia el jugador objetivo.

8. Pantalla de juego, mapas y lógica de actualización

‘GameScreen‘ concentra las responsabilidades mÆs pesadas:

? **Carga de mapas**: ‘MapManager‘ abre ‘.tmx‘, se guardan colisiones desde la capa "Capa de Objetos 1" y se extraen spawn points por capa (Player, Bolb, Slop, Fungop, Boss, Fire, Portal).
? **Gestión de entidades**: todos los actores (jugadores, enemigos, objetos) viven en ‘entities‘, mientras que ‘syncedEntities‘ rastrea lo que debe replicarse a los clientes. La actualización (‘updateEntities‘) ejecuta cooldowns, físicas y elimina entidades, generando monedas al morir enemigos.
? **Colisiones**: se resuelven de forma axis-aligned reutilizando ‘collisionObjects‘. Las cajas de daæo se comparan en ‘checkEntityDamage‘, que tambiØn abre el menœ de mejoras cuando un jugador entra en contacto con el fuego.
? **HUD y cÆmaras**: renderiza HUDs, partículas y dibuja overlays como el lÆser del jefe utilizando ‘whitePixelRegion‘. La cÆmara sigue al jugador con lerp y soporta sacudidas para impactos.
? **Progresión**: al detectar que no quedan enemigos se genera un portal en el spawn predefinido. Al tocarlo se incrementa ‘roomsClearedCount‘, se notifican clientes (‘CHANGE_LEVEL‘) y se lanza ‘InterlevelLoadingScreen‘ para cargar el siguiente mapa.
? **Sincronización**: cada frame se envía ‘UPDATE_ENTITY‘ para todos los objetos en ‘syncedEntities‘, y se responde a clientes reciØn conectados con ‘sendFullEntitySnapshotTo‘. TambiØn se manda ‘HUD_FIRE_STATE‘ individual para mantener coherencia del HUD remoto y se notifican eventos como ‘COIN_PICKED‘, ‘REMOVE_ENTITY‘ o ‘PLAYER_DIED‘.

9. Sistema de red

? **ServerThread (‘core/src/main/java/com/FK/game/network/ServerThread.java‘)** abre un ‘DatagramSocket‘ en ‘56555‘, escucha conexiones ‘CONNECT‘, asigna IDs e instancia un ‘NetworkInputHandler‘ por cliente. Un hilo secundario atiende broadcasts en ‘56556‘ para descubrimiento automÆtico.
? El servidor mantiene dos mapas: por ID y por ‘InetSocketAddress‘. Adicionalmente guarda IDs liberados para reuso.
? **Entrada remota**: mensajes ‘INPUT:<id>:<acción>‘ llegan a ‘handlePacket‘, que busca el jugador asociado y llama a ‘handleNetworkInput‘. TambiØn procesa solicitudes de upgrade, cierre

de ventana, desconexiones y reset del servidor cuando no quedan clientes.
? **Loop de replicación**: ‘gameLogicLoop‘ recorre todos los clientes ~30 veces por segundo y envía su ‘buildPlayerUpdateMessage‘. ‘GameScreen‘ complementa esto generando ‘CREATE_ENTITY/UPDATE_ENTITY‘ para enemigos, monedas, portal y fuego.
? **Mensajería**: los enums ‘NetworkMessage‘, ‘StateMessage‘, ‘EntityTypeMessage‘ definen un vocabulario claro. Aunque el protocolo usa strings simples, cada mensaje incluye ID, tipo, posición y estado, lo que facilita la reconstrucción del mundo en el cliente.
? **Manejo de emergencias**: ‘MainGame‘ y ‘ServerDisconnectWindow‘ permiten apagar el servidor de forma ordenada (broadcast de ‘SERVER_SHUTDOWN‘, reinicio de pantallas).

10. Progresión, economía y mejoras

? Las monedas (‘Coin‘) persiguen al jugador mÆs cercano, notifican al cliente cuando son recogidas y alimentan ‘PlayerData.coinCount‘.
? El fuego/hoguera (‘Fire‘) actœa como nodo social: cuando un jugador entra en su ‘DamageBox‘ se muestra el menœ de mejoras, se congelan los controles y ‘UpgradeWindow‘ consulta costos al servidor.
? ‘roomsClearedCount‘ decide cuÆndo se elige una sala del jefe; al morir un jugador se reinician estadísticas y se vuelve al mapa inicial.

11. Audio y efectos

? ‘SoundCache‘ gestiona loops (pasos, fuego) y reproducción espacial basÆndose en la distancia al jugador activo.
? ‘activeEffects‘ en ‘GameScreen‘ permite reproducir partículas (por ejemplo impactos en el suelo) y el renderizado especial de ataques de jefe.

12. Recomendaciones para mantenimiento

? Centralizar la serialización de mensajes en un helper para reducir strings "magic" repartidas entre ‘GameScreen‘ y ‘ServerThread‘.
? Documentar las capas esperadas en los mapas Tiled para evitar errores de nombre.
? Aæadir pruebas automÆticas o herramientas de validación para ‘PlayerData‘/‘UpgradeManager‘, ya que la economía crece linealmente y podría necesitar ajustes.