🔄 CICLO DE VIDA DOS REGISTRIES

📋 Resumo Executivo

Sistema de gerenciamento multiplayer com 3 registros principais:

- **PlayerRegistry**: Gerencia jogadores conectados e seus inventários
- **RoomRegistry**: Gerencia salas de lobby e histórico
- **RoundRegistry**: Gerencia partidas ativas

🚀 INICIALIZAÇÃO (ServerManager)

gdscript

```
# No ServerManager._ready() ou initialize_server()

# 1. Criar instâncias
player_registry = PlayerRegistry.new()
room_registry = RoomRegistry.new()
round_registry = RoundRegistry.new()

add_child(player_registry)
add_child(room_registry)
add_child(round_registry)

# 2. Injetar dependências (referências cruzadas)
player_registry.room_registry = room_registry
player_registry.round_registry = round_registry
player_registry.object_spawner = object_spawner

room_registry.player_registry = player_registry
room_registry.round_registry = round_registry
room_registry.object_spawner = object_spawner

round_registry.player_registry = player_registry
round_registry.room_registry = room_registry
round_registry.object_spawner = object_spawner

# 3. Inicializar
player_registry.initialize()
room_registry.initialize()
round_registry.initialize()
```

🔌 CONEXÃO DE JOGADOR

gdscript

```gdscript
# Quando cliente conecta ao servidor
func _on_peer_connected(peer_id: int):
    # 1. Adiciona ao PlayerRegistry
    player_registry.add_peer(peer_id)
```

📝 REGISTRO DE JOGADOR

gdscript

```gdscript
# Quando jogador envia nome
@rpc("any_peer", "call_remote", "reliable")
func register_player(player_name: String):
    var peer_id = multiplayer.get_remote_sender_id()

    # Registra nome
    if player_registry.register_player(peer_id, player_name):
        # Sucesso - notifica cliente
        rpc_id(peer_id, "registration_success", player_name)
        # Pode enviar lista de salas aqui
    else:
        # Falha (nome duplicado)
        rpc_id(peer_id, "registration_failed", "Nome já está em uso")
```

🏠 ENTRAR EM SALA

gdscript

```
# Cliente solicita entrar/criar sala
@rpc("any_peer", "call_remote", "reliable")
func join_room(room_id: int, password: String = ""):
    var peer_id = multiplayer.get_remote_sender_id()

    # Valida se sala existe
    if not room_registry.room_exists(room_id):
        rpc_id(peer_id, "join_failed", "Sala não existe")
        return

    # Valida senha
    var room = room_registry.get_room(room_id)
    if room["has_password"] and room["password"] != password:
        rpc_id(peer_id, "join_failed", "Senha incorreta")
        return

    # Adiciona à sala (já atualiza PlayerRegistry automaticamente)
    if room_registry.add_player_to_room(room_id, peer_id):
        # Notifica todos na sala
        _notify_room_update(room_id)
    else:
        rpc_id(peer_id, "join_failed", "Sala cheia ou em jogo")

@rpc("any_peer", "call_remote", "reliable")
func create_room(room_name: String, password: String, max_players: int):
    var peer_id = multiplayer.get_remote_sender_id()

    # Verifica nome duplicado
    if room_registry.room_name_exists(room_name):
        rpc_id(peer_id, "create_failed", "Nome já existe")
        return

    # Cria sala
    var room_id = room_registry.get_next_room_id()
    var room_data = room_registry.create_room(
        room_id,
        room_name,
        password,
        peer_id,  # host
        2,  # min_players
        max_players
    )

    if room_data:
        rpc_id(peer_id, "room_created", room_data)
```

```
    # Atualiza lista para todos
    _broadcast_room_list()
```

🎮 INICIAR PARTIDA (Rodada)

gdscript

```
# Host solicita iniciar partida
@rpc("any_peer", "call_remote", "reliable")
func start_match():
    var peer_id = multiplayer.get_remote_sender_id()

    # Pega sala do jogador
    var room = room_registry.get_player_room(peer_id)
    if room.is_empty():
        return

    # Valida se é host
    if not room_registry.is_player_host(peer_id, room["id"]):
        rpc_id(peer_id, "start_failed", "Apenas host pode iniciar")
        return

    # Valida requisitos
    if not room_registry.can_start_match(room["id"]):
        var req = room_registry.get_match_requirements(room["id"])
        rpc_id(peer_id, "start_failed", "Faltam %d jogadores" % req["missing_players"])
        return

    # Marca sala como em jogo
    room_registry.set_room_in_game(room["id"], true)

    # Cria rodada
    var round_data = round_registry.create_round(
        room["id"],
        room["name"],
        room["players"],
        room["settings"]
    )

    if round_data.is_empty():
        return

    # Carrega cena de jogo para todos na sala
    _load_game_scene_for_room(room["id"], round_data)

func _load_game_scene_for_room(room_id: int, round_data: Dictionary):
    var players = room_registry.get_room(room_id)["players"]

    for player in players:
        # Notifica cada jogador para carregar cena
        rpc_id(player["id"], "load_game_scene", round_data)

    # Aguarda todos carregarem (implementar sistema de confirmação)
```

```
    # Quando todos confirmarem, chamar:
    # _spawn_all_players_and_start_round(round_data["round_id"])

func _spawn_all_players_and_start_round(round_id: int):
    var round_data = round_registry.get_round(round_id)

    # Spawna cada jogador na cena
    for player in round_data["players"]:
        var player_node = _spawn_player(player["id"], round_data)
        # Registra node no RoundRegistry
        round_registry.register_spawned_player(round_id, player["id"], player_node)

    # INICIA a rodada (ativa timers)
    round_registry.start_round(round_id)
```

🏁 FINALIZAR PARTIDA

gdscript

```
# Quando condição de vitória é atingida
func _check_win_condition(round_id: int):
    # Determina vencedor
    var leaderboard = round_registry.get_leaderboard(round_id)
    var winner = leaderboard[0] if leaderboard.size() > 0 else {}

    # Finaliza rodada
    var final_data = round_registry.end_round(round_id, "completed", winner)

    # Mostra resultados para todos
    _show_results_to_players(final_data)

    # Aguarda tempo de exibição
    await get_tree().create_timer(round_registry.results_display_time).timeout

    # Completa finalização (adiciona ao histórico da sala)
    round_registry.complete_round_end(round_id)

    # Volta para lobby
    _return_room_to_lobby(final_data["room_id"])

func _return_room_to_lobby(room_id: int):
    # Marca sala como fora de jogo
    room_registry.set_room_in_game(room_id, false)

    # Notifica jogadores para voltar ao lobby
    var players = room_registry.get_room(room_id)["players"]
    for player in players:
        rpc_id(player["id"], "return_to_lobby", room_id)
```

🔌 DESCONEXÃO DE JOGADOR

gdscript

```gdscript
func _on_peer_disconnected(peer_id: int):
    # 1. Se estava em rodada, marca desconexão
    if player_registry.in_round(peer_id):
        var round_id = player_registry.get_player_round(peer_id)
        round_registry.mark_player_disconnected(round_id, peer_id)
        # O RoundRegistry vai verificar automaticamente se todos desconectaram

    # 2. Se estava em sala, remove da sala
    if player_registry.in_room(peer_id):
        var room_id = player_registry.get_player_room(peer_id)
        room_registry.remove_player_from_room(room_id, peer_id)
        # Sala pode ser deletada automaticamente se ficou vazia

    # 3. Remove do PlayerRegistry (limpa tudo)
    player_registry.remove_peer(peer_id)
```

📊 INVENTÁRIO (Durante Rodada)

gdscript

```
# Jogador coleta item
func player_collect_item(peer_id: int, item_name: String):
    var round_id = player_registry.get_player_round(peer_id)

    if round_id == -1:
        return

    # Adiciona ao inventário
    if player_registry.add_item_to_inventory(round_id, peer_id, item_name):
        # Sucesso
        rpc_id(peer_id, "item_added", item_name)
    else:
        # Inventário cheio
        rpc_id(peer_id, "inventory_full")

# Jogador equipa item
func player_equip_item(peer_id: int, item_name: String, slot: String):
    var round_id = player_registry.get_player_round(peer_id)

    if round_id == -1:
        return

    if player_registry.equip_item(round_id, peer_id, item_name, slot):
        # Notifica todos para atualizar visual
        _broadcast_to_round(round_id, "player_equipped_item", {
            "peer_id": peer_id,
            "item": item_name,
            "slot": slot
        })
```

🧹 SHUTDOWN COMPLETO

gdscript

```
func shutdown_server():
    # 1. Finaliza todas as rodadas ativas
    for round_id in round_registry.get_all_rounds().keys():
        round_registry.end_round(round_id, "server_shutdown")
        round_registry.complete_round_end(round_id)

    # 2. Remove todas as salas
    for room_id in room_registry.get_rooms_list():
        room_registry.remove_room(room_id["id"])

    # 3. Desconecta todos os jogadores
    for peer_id in player_registry.get_all_players():
        multiplayer.disconnect_peer(peer_id["id"])

    # 4. Reseta registries
    round_registry.reset()
    room_registry.reset()
    player_registry.reset()
```

🔍 QUERIES ÚTEIS

gdscript

```
# Onde está um jogador?
var in_room = player_registry.in_room(peer_id)
var in_round = player_registry.in_round(peer_id)
var room_id = player_registry.get_player_room(peer_id)
var round_id = player_registry.get_player_round(peer_id)

# Informações da sala
var room = room_registry.get_room(room_id)
var can_start = room_registry.can_start_match(room_id)
var stats = room_registry.get_room_statistics(room_id)

# Informações da rodada
var round = round_registry.get_round(round_id)
var state = round_registry.get_round_state(round_id)
var time_left = round_registry.get_time_remaining(round_id)
var leaderboard = round_registry.get_leaderboard(round_id)
var active_players = round_registry.get_active_players(round_id)

# Inventário
var items = player_registry.get_inventory_items(round_id, peer_id)
var equipped = player_registry.get_equipped_items(round_id, peer_id)
var has_sword = player_registry.has_item(round_id, peer_id, "sword")
```

⚠️ PONTOS CRÍTICOS

1. Ordem de Inicialização

- Sempre criar → injetar dependências → inicializar

2. Limpeza Automática

- RoomRegistry remove sala vazia automaticamente
- RoundRegistry limpa rodada após `complete_round_end()`
- PlayerRegistry limpa inventários ao sair de rodada

3. Estados Circulares

```
JOGADOR: conectado → registrado → em_sala → em_rodada → em_sala → desconectado
SALA: criada → lobby → em_jogo → lobby → (vazia = deletada)
RODADA: loading → playing → ending → results → (deletada)
```

4. Sincronização

- PlayerRegistry atualiza automaticamente quando:
  - Entra/sai de sala via RoomRegistry

- Entra/sai de rodada via RoundRegistry

- RoomRegistry adiciona rodada ao histórico automaticamente

5. Verificações Automáticas

- RoundRegistry verifica desconexões a cada 2s
- Se todos desconectam, finaliza rodada automaticamente
- Timer de duração máxima finaliza rodada por timeout

📝 CHECKLIST DE IMPLEMENTAÇÃO

- [ ] Criar instâncias dos 3 registries no ServerManager
- [ ] Injetar dependências cruzadas
- [ ] Conectar sinais de multiplayer (`peer_connected`, `peer_disconnected`)
- [ ] Implementar RPCs de registro, sala e rodada
- [ ] Implementar sistema de spawn de jogadores
- [ ] Implementar lógica de vitória/derrota
- [ ] Implementar sistema de inventário
- [ ] Testar ciclo completo: conectar → criar sala → jogar → desconectar
- [ ] Testar desconexões durante rodada
- [ ] Testar sala ficando vazia

🎯 BENEFÍCIOS DA ARQUITETURA

✅ **Separação de responsabilidades** - Cada registry tem função única ✅ **Circular completo** - Tudo que é criado pode ser destruído ✅ **Histórico preservado** - Salas mantêm estatísticas de rodadas passadas ✅ **Queries rápidas** - `player.in_room()`, `player.in_round()` ✅ **Limpeza automática** - Recursos liberados corretamente ✅ **Escalável** - Múltiplas salas e rodadas simultâneas ✅ **Debug facilitado** - `debug_print_all_*()` em cada registry