PROFESSOR: Adriano Felix Valente	
CURSO: Ciência da Computação	
DISCIPLINA: Algoritmos e Lógica de Programação	
TURMA: Matutino	DATA: 18/05/2025
ALUNOS: Analice Coimbra Carneiro, Harry Zhu, João Pedro Da Silva, Rafaela Florêncio Morais	

TÍTULO DA ATIVIDADE:

Análise da Aplicação de Algoritmos com estruturas de funções e procedimentos

Serão analisados os scripts “KeepAudioPlaying.cs”, “LoreManager1.cs”, “MonitorEventSystem.cs”, “MonsterBehavior.cs”, “PowerEnergy.cs” e “Radio.cs” identificando o uso dessas estruturas algorítmicas e seus resultados dentro do contexto do projeto.

Estruturas de Decisão

Estruturas de decisão permitem que o programa execute diferentes blocos de código com base em uma condição. Nos scripts fornecidos, a estrutura de decisão fundamental utilizada é o if, frequentemente acompanhado de else if ou else, e também verificações de nulidade para evitar erros.

KeepAudioPlaying.cs: Este script utiliza uma estrutura de decisão no método `OnSceneLoaded`, que é chamado sempre que uma nova cena é carregada. Um `if` verifica se o índice de build da cena carregada (`scene.buildIndex`) é igual à variável `cenaOndeDestruir`. Se essa condição for `true`, uma mensagem é logada no console indicando que o áudio está sendo destruído (`Debug.Log($" '{cenaOndeDestruir}' Destruindo audio");`) e o objeto de jogo associado a este script é destruído (`Destroy(this.gameObject);`). O resultado dessa decisão é a persistência do áudio entre cenas, exceto na cena especificada pelo índice `cenaOndeDestruir`, onde ele é explicitamente removido.

LoreManager1.cs: Nos trechos fornecidos, este script `LoreManager1` possui métodos para carregar cenas (`StartStory` e `StartPlay`) utilizando `SceneManager.LoadScene`. No entanto, não há estruturas de decisão (`if`, `else if`, `else`) presentes nesses métodos ou em outras partes visíveis do código. As ações de carregar cenas são executadas diretamente quando os métodos são chamados.

MonitorEventSystem.cs: O script `MonitorEventSystem` faz uso extensivo de estruturas de decisão `if` e `else if` no método `Update` para monitorar o status de vários elementos do jogo e atualizar a interface gráfica (`TextMeshPro`) de acordo. Primeiro, ele verifica o status do gerador com `if (powerEnergyScript.geradorQuebrado)`. Se a variável `geradorQuebrado` for `true`, o texto de status do gerador (`generatorStatusText.text`) é definido como `"Generator_status >> ERROR"` e sua cor é alterada para vermelho (`Color.red`). Caso contrário, um `else if (!powerEnergyScript.geradorQuebrado)` verifica se o gerador NÃO está quebrado; se for `true`, o texto é definido como `"Generator_status >> OK"` e a cor para

verde (`Color.green`). O resultado é uma representação visual clara do estado do gerador na UI. Similarmente, ele verifica o status das câmeras (`cam1`, `cam2`, `cam3`) com `if (cam1.spotted || cam2.spotted || cam3.spotted)`. Se qualquer uma das câmeras tiver a variável `spotted` como `true`, o texto de status da câmera (`cameraStatusText.text`) se torna `"Camera_status >> LOOK"` e a cor vermelha. Um `else if (!cam1.spotted && !cam2.spotted && !cam3.spotted)` verifica se NENHUMA câmera detectou algo; se verdadeiro, o texto é `"Camera_status >> OK"` e a cor verde. Isso reflete na UI se há movimento detectado pelas câmeras. Por fim, ele verifica o status dos sensores de movimento (`movSensor1`, `movSensor2`, `movSensor3`) com `if (movSensor1.spotted || movSensor2.spotted || movSensor3.spotted)`. Se qualquer sensor detectar movimento (`spotted` for `true`), o texto de status de movimento (`movementStatusText.text`) é `"Movement_status >> LOOK"` e a cor vermelha. O `else if (!movSensor1.spotted && !movSensor2.spotted && !movSensor3.spotted)` cobre o caso em que nenhum sensor detectou movimento, definindo o texto para `"Movement_status >> OK"` e a cor para verde. O resultado é uma indicação visual na UI da detecção de movimento pelos sensores.

MonsterBehavior.cs: Este script utiliza estruturas de decisão no método `OnMouseOver` para determinar a interação com o monstro. Um `if` verifica se o item ativo do jogador (`itemSwitcher.activeItem`) é igual a `1` E se a distância entre o monstro e o jogador (`Vector3.Distance(transform.position, player.position)`) é menor que a `detectionDistance`. Se ambas as condições forem verdadeiras, a UI de interação é ativada e os textos de ação e comando são definidos. Dentro deste `if`, há outro `if (Input.GetKeyDown(KeyCode.F))` que verifica se a tecla 'F' (`KeyCode.F`) foi pressionada. Se a tecla 'F' for pressionada (enquanto as condições externas são verdadeiras), os textos da UI são limpos, a UI é desativada e o método `ReturnHiding` é chamado, fazendo o monstro voltar para sua localização de esconderijo. O resultado é uma interação condicional baseada na proximidade do jogador, no item equipado e na ação do jogador. O método `IsActive` utiliza `Random.Range` para selecionar uma destinação aleatória, o que é uma forma de aleatoriedade, mas não uma estrutura de decisão `if`/`else if` baseada em condições booleanas nesse trecho.

PowerEnergy.cs: O script `PowerEnergy` emprega diversas estruturas de decisão para gerenciar o estado do gerador e seus efeitos no ambiente.

No método `Update`, um if `(!geradorQuebrado)` verifica se o gerador NÃO está quebrado. Se for true, a vida do gerador (vidaGerador) diminui a cada frame. Dentro deste if, outro if `(vidaGerador <= 0f)` verifica se a vida do gerador chegou a zero ou menos. Se esta condição for verdadeira (e o gerador não estava quebrado), o gerador é marcado como quebrado `(geradorQuebrado = true)`, o método `DesligarLuzes` é chamado, o som de queda de energia `(powerOutage)` é reproduzido, e uma mensagem de depuração é logada. O resultado é a automação da falha do gerador com o tempo, desencadeando a desativação de luzes, câmeras e sensores.

No método `OnMouseOver`, um `if (geradorQuebrado && distancia <= distanciaParaSegurar)` verifica se o gerador ESTÁ quebrado E se o jogador está próximo o suficiente. Se ambas as condições forem verdadeiras, a UI de interação para consertar é ativada. Dentro deste bloco, outro `if (Input.GetKey(KeyCode.E))` verifica se a tecla 'E' está sendo pressionada. Se for, o tempo segurando (tempoSegurando) aumenta. Um `if (tempoSegurando >= tempoParaReparo)` verifica se o tempo segurando atingiu o necessário para o reparo. Se

verdadeiro, o método `ConsertarGerador` é chamado, o tempo segurando é resetado, a UI é desativada e o som de reparo para. Um else associado ao `if (Input.GetKey(KeyCode.E))` reseta o `tempoSegurando` e para o som se a tecla 'E' for solta antes do tempo total. O resultado é um mecanismo de reparo interativo que requer que o jogador segure a tecla 'E' por um tempo determinado enquanto está perto do gerador quebrado.

Os métodos `DesligarLuze`s e `LigarLuzes` utilizam loops foreach para iterar sobre arrays de objetos com tags específicas (precisaLuz para "Light", sensores para "sensor"). Dentro desses loops, estruturas if são usadas para verificar se os componentes `(Light, Camera, MonoBehaviour, Collider)` foram encontrados nos objetos e, no caso das câmeras, se a RenderTexture não é nula e se a textura original já foi salva no dicionário. Essas decisões garantem que apenas componentes válidos sejam habilitados/desabilitados ou manipulados, e que a RenderTexture original seja restaurada corretamente ao ligar as luzes.

Radio.cs: Este script emprega estruturas de decisão para controlar a visibilidade da UI do rádio e iniciar o diálogo.

No método `Start`, um `if (RadioUI != null)` verifica se a referência ao objeto da UI do rádio não é nula. Se for válida, a UI é desativada inicialmente `(RadioUI.SetActive(false);)`. O resultado é garantir que a UI do rádio esteja oculta ao iniciar.

No método OnTriggerEnter, que é chamado quando um colisor entra no trigger do rádio, um if `(other.CompareTag("Player") && !dialogoIniciado)` verifica se o objeto que entrou tem a tag "Player" E se o diálogo ainda não foi iniciado (!dialogoIniciado). Se ambas as condições forem verdadeiras, a variável dialogoIniciado é definida como true, um método na barraSanidade é chamado, e a corrotina `ExecutarDialogo` é iniciada. O resultado é que o diálogo do rádio é ativado apenas uma vez, quando o jogador entra na área do trigger pela primeira vez.

BarradeSanidade.cs: Este script faz uso de estruturas de decisão `if` e `else` para gerenciar a sanidade do jogador, atualizar a barra de UI associada e determinar condições de fim de jogo. No método `Update`, diversas verificações condicionais ocorrem a cada frame.

Um `if (sanidadeAtual != lastSanidade)` verifica se o valor da sanidade foi alterado, e se sim, inicia ou continua o processo de interpolação (`Lerp`) para suavizar a atualização visual da barra de sanidade na UI. Dentro deste bloco, um `if (lerpTimer >= chipSpeed)` determina se a interpolação atingiu o tempo definido, finalizando a animação e resetando variáveis de controle.

Outro `if (monsterBehavior != null && geradorQuebrado != null)` realiza uma verificação para garantir que as referências a outros scripts (`MonsterBehavior` e `PowerEnergy`) são válidas antes de tentar acessar suas propriedades. Dentro deste bloco, um `if (geradorQuebrado.geradorQuebrado)` e seu `else` associado decidem se a sanidade deve diminuir a uma alta taxa (gerador quebrado) ou aumentar a uma taxa menor (gerador funcionando).

Outro `if (!monsterBehavior.monsterHiding)` verifica se o monstro *não* está escondido, aplicando uma perda adicional de sanidade caso verdadeiro.

Por fim, um `if (mudancaSanidadeNesteFrame != 0f)` verifica se houve qualquer alteração na sanidade neste frame antes de chamar o método `AlterarSanidade`. No método `AlterarSanidade`, que é responsável por aplicar a mudança de valor e garantir que a sanidade permaneça dentro dos limites (usando `Mathf.Clamp`), um `if (sanidadeAtual <= 0)` verifica se a sanidade caiu para zero ou menos. Se essa condição for verdadeira, a cena "YouDied" é carregada, resultando no fim do jogo.