Documentação do Backend do Projeto

**Nota:** Este documento ainda está em construção e é atualizado conforme a lógica do projeto é implementada.

Classes e suas Funções

BattleManager: A classe BattleManager é responsável por gerenciar o ciclo de vida das batalhas, organizando a sequência de fases e prevenindo a sobreposição de eventos. Ela mantém o controle sobre as ações dos Digimons, seus treinadores (tamers), e o tempo necessário para cada ação, além de controlar os turnos e as fases da batalha. A classe também verifica se há triggers ativos para eventos como início de turno, fim de turno, e início da batalha.

**Propriedades:**

* **player: Player**: Referência ao jogador.
* **enemy: Enemy**: Referência ao inimigo.
* **playerDigimon: Digimon**: Digimon do jogador.
* **enemyDigimon: Digimon**: Digimon do inimigo.
* **changeTurnTimer: Timer**: Timer para controlar a mudança de turno.
* **BM: BattleMessenger**: Referência para exibir mensagens durante a batalha.

**Variáveis de controle:**

* **turn: int**: Controla o número do turno atual.
* **currentPhase: Enums.BattlePhase**: A fase atual da batalha (início, escolha de ação, etc.).
* **somethingIsHappening: int**: Contador de eventos simultâneos acontecendo na batalha.
* **currentDigimon: Digimon**: O Digimon que está realizando a ação no momento.
* **oppositeDigimon: Digimon**: O Digimon adversário do que está agindo.
* **currentTamer: Tamer**: O treinador do Digimon atual.
* **oppositeTamer: Tamer**: O treinador do Digimon adversário.
* **tamerReady: int**: Controla o número de treinadores prontos.
* **choosing: bool**: Controla se o jogador está escolhendo uma ação.
* **isPassing: bool**: Controla se o turno está sendo passado diretamente.

**Métodos principais:**

* **inAction()**: Incrementa o contador de eventos ativos e ajusta o timer.
* **outAction(location: String)**: Decrementa o contador de eventos e realiza verificações para continuar o fluxo da batalha.
* **changeTurn()**: Controla a lógica de mudança de turno, verificando fases e executando triggers associados a elas.
* **generateActions(actor: Digimon, opponent: Digimon) -> int**: Calcula o número de ações de um Digimon em relação à sua velocidade (agilidade) e à do oponente.
* **triggerCheck(triggers: Array, digimon: Digimon, context)**: Verifica e executa triggers ativos para o Digimon dado um contexto específico.
* **changeActor()**: Alterna entre os Digimons (jogador e inimigo) para definir quem está agindo.
* **gettingStarted()**: Inicializa o início da batalha, configurando as variáveis de turno.
* **passingTurn()**: Lógica para passar o turno de um Digimon sem realizar ações.
* **manageTimer()**: Garante que o timer de mudança de turno esteja funcionando corretamente.

BattleMesenger: A classe BattleMessenger é responsável por exibir mensagens na tela durante a batalha. Ela garante que as mensagens não se sobreponham, mantendo um fluxo ordenado e ritmado de exibição. Também interage com a classe BattleManager para pausar o fluxo de eventos da batalha enquanto a mensagem é exibida, evitando que mensagens importantes sejam ignoradas.

**Propriedades:**

* **speaking: bool**: Variável de controle para saber se uma mensagem está sendo exibida no momento.
* **messageQueue: Array[String]**: Fila de mensagens que aguardam para ser exibidas.
* **changeMessageTimer: Timer**: Temporizador que controla o tempo entre a exibição de cada mensagem.
* **content: RichTextLabel**: O componente que exibe o conteúdo da mensagem na tela.
* **BTM: BattleManager**: Referência ao **BattleManager**, usado para pausar e retomar o fluxo da batalha.

**Métodos principais:**

* **showMessage(newMessage: String)**: Adiciona uma nova mensagem à fila. Se não há outra mensagem sendo exibida, começa o processo de exibição.
* **speak()**: Exibe a próxima mensagem na fila. Se houver mais mensagens, continua exibindo-as com um intervalo.
* **onChangeMessageTimeout()**: Controla o fim do tempo de exibição de uma mensagem. Se há mais mensagens na fila, chama speak() para exibir a próxima; caso contrário, retoma o fluxo da batalha.
* **changeText(newMsg: String)**: Atualiza o conteúdo da mensagem no componente visual (RichTextLabel), garantindo que o texto seja formatado e exibido corretamente.

ButtonDescription: A classe ButtonDescription é uma classe simples, projetada para modificar a aparência e funcionalidade de um botão, geralmente associado a habilidades. Sua principal função é associar um botão a uma habilidade específica e atualizar suas propriedades visuais e de usabilidade conforme o estado da habilidade muda (por exemplo, se está disponível ou em cooldown).

**Propriedades:**

* **buttonName: String**: Nome exibido do botão.
* **buttonDescription: String**: Descrição da habilidade associada ao botão.
* **isSkillButton: bool**: Indica se o botão representa uma habilidade.
* **relatedSkill: Skill**: A habilidade relacionada ao botão.

**Métodos principais:**

* **associateSkill(skill: Skill)**: Associa uma habilidade ao botão, configurando o nome, descrição e ícone do botão de acordo com a habilidade.
* **updateSkills()**: Atualiza o estado do botão de acordo com o estado atual da habilidade (por exemplo, se a habilidade está em cooldown ou se pode ser usada). Desabilita o botão se a habilidade não for usável e exibe o cooldown, se houver.

ButtonPanel: A classe ButtonPanel gerencia todos os botões visíveis na tela para o jogador durante a batalha. Ela é responsável por configurar, atualizar e controlar a interação do jogador com os botões, seja para escolher ações, acessar menus, ou passar o turno. A classe também manipula o fluxo de ação na batalha, garantindo que apenas botões válidos sejam ativados, exibindo descrições dos botões e integrando-se com o sistema de batalha.

**Propriedades:**

* **relatedPlayer: Player**: Referência ao jogador associado.
* **relatedDigimon: Digimon**: Referência ao Digimon associado.
* **allButtons: Array[ButtonDescription]**: Lista de todos os botões no painel.
* **skillButtons: Array[ButtonDescription]**: Lista de botões associados às habilidades do Digimon.
* **showDescTimer: Timer**: Temporizador para controlar a exibição de descrições dos botões.
* **infoWindow: InfoWindow**: Janela de informações que exibe a descrição de um botão.
* **BM: BattleMessenger**: Responsável por exibir mensagens durante a batalha.
* **BTM: BattleManager**: Gerencia o fluxo de batalha e controla as fases.
* **descWindowOn: bool**: Indica se a janela de descrição está ativa.
* **currentButtonToShow: ButtonDescription**: O botão atual cuja descrição será exibida.

**Métodos principais:**

* **\_ready()**: Define os botões padrões na interface, como o botão de passar o turno e o de inventário.
* **setButtons()**: Associa os botões de habilidades às habilidades do Digimon.
* **updateButtons()**: Atualiza o estado dos botões de habilidades, verificando se as habilidades podem ser usadas ou estão em cooldown.
* **blockAllButtons()**: Torna todos os botões invisíveis, bloqueando a interação.
* **unBlockAllButtons()**: Torna todos os botões visíveis novamente e atualiza o estado de cada um.
* **showInfoWindow()**: Exibe a janela de informações com a descrição do botão selecionado.
* **activateButton()**: Valida e ativa o botão selecionado, garantindo que a habilidade possa ser usada e disparando a ação do Digimon, se válida.

**Métodos de interação com os botões:**

Cada um desses métodos é chamado quando um botão é pressionado:

* **\_on\_pass\_turn\_button\_down()**
* **\_on\_menu\_button\_down()**
* **\_on\_inventory\_button\_down()**
* **\_on\_companion\_button\_down()**
* **\_on\_basic\_atack\_button\_down()**
* **\_on\_signature\_skill\_button\_down()**
* **\_on\_extra\_skill\_1\_button\_down()**
* **\_on\_extra\_skill\_2\_button\_down()**
* **\_on\_extra\_skill\_3\_button\_down()**

DamageData: A classe DamageData é usada de forma dinâmica para armazenar e manipular dados de dano quando ele é recebido ou causado por um Digimon. Ela não é usada de forma estática, mas é criada conforme necessário durante as batalhas para fornecer uma estrutura que descreve o tipo de dano, seu valor, e outras características importantes, como elemento e se o ataque foi crítico. Esses dados são então usados nas funções que calculam e aplicam o dano aos personagens.

**Propriedades:**

* **damageType: Enums.DamageType**: Tipo principal do dano (físico, mágico, etc.).
* **damageSubType: Enums.DamageSubType**: Subtipo do dano, detalhando mais sua natureza.
* **damageElement: Enums.Element**: O elemento associado ao dano (fogo, água, etc.).
* **atackerType: Enums.Type**: O tipo do atacante (pode ser usado para determinar vantagens de tipo).
* **damageValue: float**: O valor numérico do dano causado.
* **isCritic: bool**: Indica se o ataque foi um crítico.

**Método principal:**

* **buildData()**: Método que constrói o objeto **DamageData**, recebendo como parâmetros todos os dados necessários (tipo, subtipo, elemento, tipo do atacante, valor do dano e se foi um ataque crítico). Esses dados são então armazenados nas respectivas variáveis da classe para uso posterior.

DamageSkill: A classe DamageSkill é uma subclasse de Skill e representa habilidades que causam dano no jogo. Ela lida com a ativação de habilidades de dano, o cálculo dos valores de dano com base nos atributos do Digimon e a aplicação de efeitos críticos e outros modificadores. Além disso, transfere dados estáticos sobre a habilidade de uma estrutura de dados externa e manipula o estado dinâmico do Digimon para calcular o dano final.

**Propriedades:**

* **ratio: float**: Multiplicador usado no cálculo de dano.
* **isCritic: bool**: Indica se o ataque é crítico (true/false).
* **damageValue: float**: Valor final do dano a ser causado.
* **hasEffect: bool**: Indica se a habilidade tem um efeito adicional além do dano.
* **element: Enums.Element**: O elemento da habilidade (fogo, água, etc.).
* **atackerType: Enums.Type**: Tipo do Digimon atacante.
* **damageType: Enums.DamageType**: Tipo do dano causado (físico, mágico, etc.).
* **damageSubType: Enums.DamageSubType**: Subtipo do dano causado.

**Métodos principais:**

* **setStats(stats: DamageSkillData)**: Carrega os dados estáticos da habilidade a partir de um objeto **DamageSkillData**, como o ícone da habilidade, descrição, prioridade, elemento, custo de mana, e cooldown.
* **setValues(digimon: Digimon)**: Configura os valores dinâmicos da habilidade com base no Digimon atacante, como o tipo do atacante, dano base, precisão e chance de crítico. Multiplica o dano pela **ratio** e, se o ataque for crítico, aumenta o dano em 50%.
* **effect(digimon: Digimon)**: Executa a habilidade de dano, aplicando o cooldown e chamando a função **getTageted** no inimigo. Atualiza os botões de habilidades no painel do jogador, se o jogador for o atacante.
* **applyStats(\_digimon: Digimon)**: Um método vazio (placeholder) que pode ser usado para aplicar modificadores de atributos no futuro, se a habilidade tiver efeitos adicionais.
* **skillSingularity(\_digimon: Digimon)**: Um método abstrato para executar comandos específicos da habilidade, caso haja algum comportamento exclusivo para certas habilidades de dano.

DamageSkillData: A classe DamageSkillData é um recurso utilizado para armazenar dados de habilidades de dano no banco de dados do jogo. Ela contém informações estáticas e predefinidas sobre habilidades, como seus atributos, ícone, efeitos visuais, custo de mana, e cooldown. Essas informações são usadas por classes como DamageSkill para configurar e aplicar as habilidades de dano durante o jogo.

**Propriedades:**

* **skillId: int**: Identificador único da habilidade.
* **skillIcon: CompressedTexture2D**: Ícone visual da habilidade, usado na interface do usuário.
* **staticPriority: int**: Prioridade estática da habilidade (influencia a ordem ou importância em relação a outras habilidades).
* **element: Enums.Element**: O elemento associado à habilidade (fogo, água, etc.).
* **damageType: Enums.DamageType**: Tipo de dano causado pela habilidade (físico, mágico, etc.).
* **damageSubType: Enums.DamageSubType**: Subtipo de dano, detalhando ainda mais a natureza do dano.
* **ratio: float**: Multiplicador usado no cálculo do dano da habilidade.
* **manaCost: int**: Custo de mana necessário para ativar a habilidade.
* **coolDown: int**: Tempo de recarga (cooldown) da habilidade em turnos.
* **hasEffect: bool**: Indica se a habilidade possui efeitos adicionais além do dano.
* **skillTexture: CompressedTexture2D**: Textura associada aos efeitos visuais da habilidade.
* **textureRange: int**: Quantidade de frames que a textura possui. Esse valor é necessário para a configuração do elemento em cena que vai reproduzir o seu efeito visual.

Digimon: A classe Digimon representa os monstros controláveis no jogo. Ela contém informações detalhadas sobre os atributos, habilidades, e estado do Digimon. A classe também gerencia o cálculo de dano, defesa, cura, e o aprendizado de habilidades. Além disso, ela lida com a interação entre o Digimon e outros componentes da batalha, como triggers (gatilhos) que acontecem no início e no fim do turno, e as animações do Digimon.

**Propriedades principais:**

1. **Database Info**:
   * **digimonId: int**: ID do Digimon.
   * **digimonName: String**: Nome do Digimon.
   * **digimonIcon: CompressedTexture2D**: Ícone do Digimon.
   * **digimonDescription: String**: Descrição do Digimon.
2. **Enums**:
   * **element: Enums.Element**: Elemento do Digimon (fogo, água, etc.).
   * **digimonTier: Enums.Tier**: Tier do Digimon (nível de poder).
   * **digimonType: Enums.Type**: Tipo do Digimon.
3. **Atributos base e por nível**:
   * **baseSTR, baseINT, baseAGI, baseVIT, baseWIS, baseDEX**: Atributos base do Digimon.
   * **levelSTR, levelINT, levelAGI, levelVIT, levelWIS, levelDEX**: Atributos por nível.
   * **bonusSTR, bonusINT, bonusAGI, bonusVIT, bonusWIS, bonusDEX**: Bônus de atributos.
4. **Vida, Mana, Experiência e Nível**:
   * **maxHelth, currentHealth**: Vida máxima e atual.
   * **maxMana, currentMana**: Mana máxima e atual.
   * **currentExp, maxExp**: Experiência atual e máxima.
   * **currentLevel: int**: Nível atual do Digimon.
5. **Outros atributos de combate**:
   * **totalDamage: int**: Valor do dano calculado.
   * **criticalChance: int**: Chance de crítico.
   * **currentAccuracy: int**: Precisão do Digimon.
6. **Controle de batalha e animações**:
   * **skillSpawner: SkillSpawner**: Controla a exibição dos VFX de habilidades.
   * **enemy: Digimon**: Oponente.
   * **digimonSprite: Sprite2D**: Sprite visual do Digimon.
   * **digimonAnimator: AnimationPlayer**: Controla as animações do Digimon.
   * **tamer: Tamer**: Treinador do Digimon.
7. **Controle de habilidades e status**:
   * **digimonSkills: Array[Skill]**: Habilidades do Digimon.
   * **isDisabled, isBlind**: Status negativos do Digimon.
8. **Triggers (gatilhos)**:
   * **onBattleStart, onTurnStart, onTurnEnd**: Arrays de triggers para eventos específicos da batalha.
9. **Fila de ações e efeitos**:
   * **actionsToGo: Array[Skill]**: Ações que o Digimon ainda tem que executar.
   * **effectsToGo: Array[Trigger]**: Efeitos que ainda precisam ser processados.

**Métodos principais:**

1. **setBehave()**: Define o comportamento visual do Digimon, como a animação e a posição do sprite, dependendo de sua tier e posição na tela.
2. **setStats(stats: DigimonData)**: Inicializa as estatísticas do Digimon a partir de dados externos.
3. **getAttribute(att: String) -> int**: Retorna o valor total de um atributo, somando base e bônus.
4. **getCriticalChance() -> int**: Retorna a chance de acerto crítico do Digimon, baseada em seu atributo de destreza (DEX).
5. **getAccuracy() -> int**: Retorna a precisão do Digimon, baseada no atributo de destreza.
6. **getSkillDamage(damageType: Enums.DamageType) -> int**: Calcula o dano de uma habilidade, dependendo do tipo de dano (físico ou mágico).
7. **getTageted(skill: Skill)**: Executa a habilidade do inimigo contra o Digimon.
8. **gotTargeted(skill: Skill)**: Verifica se o ataque foi bem-sucedido e processa o dano se o ataque acertou.
9. **processDamage(damageData: DamageData)**: Aplica o dano ao Digimon, levando em consideração resistências de tipo e elementos.
10. **learnSkill(skill: Skill)**: Permite que o Digimon aprenda uma nova habilidade, se houver espaço.
11. **levelUpAttributes(level: int)**: Calcula o aumento de atributos ao subir de nível e ajusta a vida e mana máximas.
12. **heal(value: float, isMana: bool)**: Cura o Digimon, seja na vida ou no mana.
13. **action()**: Executa a próxima ação ou efeito na fila.
14. **animationFinished(anim\_name: String)**: Controla o que acontece após uma animação terminar.
15. **applyDefense(damageData: DamageData)**: Aplica a defesa do Digimon ao cálculo de dano recebido, reduzindo o valor final.
16. **chooseAction(newAction: Skill)**: Adiciona uma nova ação à fila de ações do Digimon.

DigimonData: A classe DigimonData é um recurso usado para armazenar os dados estáticos de um Digimon no banco de dados do jogo. Esses dados incluem identificadores únicos, atributos por nível, elementos e informações visuais, como a textura do Digimon. A classe é utilizada pela classe Digimon para inicializar suas estatísticas e aparência com base nos valores armazenados.

**Propriedades:**

1. **Identificação**:
   * **digimonId: int**: Identificador único do Digimon.
   * **digimonTier: Enums.Tier**: O tier (nível de poder) do Digimon.
   * **digimonType: Enums.Type**: Tipo do Digimon.
   * **element: Enums.Element**: Elemento associado ao Digimon.
   * **texture: CompressedTexture2D**: Textura visual do Digimon, usada para exibir o sprite.
2. **Atributos estáticos de nível**:
   * **levelSTR: int**: Valor de força ganho por nível.
   * **levelINT: int**: Valor de inteligência ganho por nível.
   * **levelAGI: int**: Valor de agilidade ganho por nível.
   * **levelVIT: int**: Valor de vitalidade ganho por nível.
   * **levelWIS: int**: Valor de sabedoria ganho por nível.
   * **levelDEX: int**: Valor de destreza ganho por nível.

DigimonDB: A classe DigimonDB é um banco de dados simples que armazena os caminhos para os arquivos DigimonData. Ela usa um dicionário para mapear IDs de Digimons a seus respectivos arquivos de recurso (.tres). Esses arquivos contêm as informações estáticas de cada Digimon, como atributos, elementos, e aparência. A classe também oferece uma função para recuperar os dados de um Digimon com base no seu índice.

**Propriedades:**

* **digimons: Dictionary**: Dicionário que armazena os caminhos para os recursos **DigimonData**. As chaves são os IDs dos Digimons, e os valores são os recursos carregados através de load().

**Métodos principais:**

* **getDigimonData(index: int) -> DigimonData**: Retorna o recurso **DigimonData** associado ao índice fornecido. O índice é a chave no dicionário que mapeia para o arquivo de recurso correspondente.

Tamer: A classe Tamer é uma classe base abstrata para controlar os Digimons. Ela não é utilizada diretamente, mas serve como base para subclasses específicas que serão controladas pelo jogador ou pela Inteligência Artificial (IA). A classe contém métodos e propriedades comuns para qualquer tamer, como o nome, o Digimon associado, o nível do tamer, e elementos de cena que controlam a interface do jogador (HUD). Ela também controla as ações durante a batalha, como a invocação de Digimons e a passagem de turnos.

**Propriedades:**

1. **Identificação**:
   * **tamerName: String**: Nome do tamer.
2. **Elementos de Cena**:
   * **digimon: Digimon**: O Digimon controlado pelo tamer.
   * **opponent: Tamer**: Oponente do tamer.
   * **HUDD: HUDDisplay**: Exibe as informações de vida, mana, e outras métricas do Digimon.
   * **judge: BattleManager**: Gerencia o fluxo da batalha.
   * **turnFrame: Sprite2D**: Sprite que mostra a moldura ao redor do tamer.
   * **frameAnimator: AnimationPlayer**: Controla a animação da moldura do tamer.
   * **BM: BattleMessenger**: Envia mensagens durante a batalha.
   * **BTM: BattleManager**: Referência ao gerenciador da batalha.
   * **numberSpawner: NumberSpawner**: Gera números na tela (como dano, cura, etc.).
3. **Atributos do Tamer**:
   * **tamerLevel: int**: Nível do tamer.
   * **actions: int**: Número de ações disponíveis no turno.
   * **canAct: bool**: Indica se o tamer pode agir no turno atual.

**Métodos principais:**

1. **summonDigimon(index: int)**: Invoca o Digimon do tamer, utilizando o índice fornecido para carregar os dados do Digimon a partir do banco de dados (**DigimonDB**).
2. **takeTurn() -> void**: Controla a passagem de turno. Por ser uma classe base, esse método é destinado a ser substituído nas subclasses específicas. Ele define o comportamento visual do tamer durante a mudança de turno.
3. **showContent(content)**: Mostra conteúdo visual na tela, utilizando o **numberSpawner** para exibir números de dano, cura ou outras informações durante a batalha.

Enemy: A classe Enemy é uma subclasse de Tamer e representa o oponente controlado pela IA (Inteligência Artificial) no jogo. Essa classe contém a lógica necessária para que a IA faça escolhas de habilidades e tome ações durante o combate, simulando um comportamento estratégico. O sistema de escolha da IA prioriza habilidades com base em sua prioridade calculada e decide passar o turno caso nenhuma habilidade válida esteja disponível.

**Propriedades:**

A classe herda todas as propriedades de **Tamer**, como o nome do tamer, o Digimon controlado, o HUD para exibir informações, e as referências para o **BattleManager** e **BattleMessenger**.

**Métodos principais:**

1. **\_ready() -> void**:
   * Inicializa o nome do tamer como "Enemy".
   * Torna invisível o **turnFrame**, que é usado para destacar o tamer do jogador.
   * Invoca aleatoriamente um Digimon a partir de um índice entre 0 e 5. (Apenas durante fase de testes)
   * O Digimon controlado pela IA aprende uma habilidade básica, **BasicAttack**.
   * Atualiza os valores do HUD.
   * Confirma ao **BattleManager** que o oponente está pronto para iniciar a batalha chamando judge.gettingStarted().
2. **makeAChoice() -> void**:
   * A lógica principal da IA para fazer escolhas de habilidades.
   * A IA percorre as habilidades do Digimon e calcula a prioridade de cada uma usando **priorityCalculation**.
   * Seleciona a habilidade com a maior prioridade. Caso duas habilidades tenham a mesma prioridade, uma é escolhida aleatoriamente com **Util.pickOne()**.
   * Se uma habilidade válida for encontrada, a IA a seleciona com **digimon.chooseAction(selectedSkill)** e finaliza a ação.
   * Se nenhuma habilidade válida for encontrada (ou a prioridade de todas for zero), a IA passa o turno.
3. **onFrameAnimation(anim\_name: String)**:
   * Um método que é chamado quando a animação de frame de turno é finalizada.
   * Exibe uma mensagem de batalha e chama **makeAChoice()** para a IA escolher uma habilidade.

Player: A classe Player é uma subclasse de Tamer que representa o jogador no jogo. Seu foco está em permitir a interação do jogador com a interface, gerenciando o painel de botões e garantindo que o jogador possa escolher as ações do Digimon de forma eficaz durante o combate. Assim como na classe Enemy, o Digimon controlado é invocado e configurado, e as habilidades são associadas e atualizadas no painel de botões.

**Propriedades:**

A classe herda todas as propriedades de **Tamer**, incluindo o **Digimon**, o HUD, o **BattleManager**, e o **BattleMessenger**. Além disso, a **Player** tem um elemento adicional específico para a interação com o jogador:

* **buttonPanel: ButtonPanel**: O painel de botões que exibe as habilidades e opções de ação disponíveis para o jogador.

**Métodos principais:**

1. **\_ready() -> void**:
   * Define o nome do tamer como "Player".
   * Torna invisível o **turnFrame**.
   * Invoca aleatoriamente um Digimon usando um índice entre 0 e 5. (apenas durante fase de testes)
   * O Digimon controlado pelo jogador aprende uma habilidade básica, **BasicAttack**.
   * Atualiza os valores do HUD, que exibe informações como vida e mana.
   * Configura o painel de botões com as habilidades do Digimon, chamando buttonPanel.setButtons() e buttonPanel.updateButtons().
   * Notifica o **BattleManager** que o jogador está pronto para iniciar a batalha, chamando judge.gettingStarted().
2. **onFrameAnimation(anim\_name: String)**:
   * Este método é chamado quando a animação do frame de turno termina.
   * Ele permite que o jogador aja, definindo canAct como true e exibindo uma mensagem de batalha através do **BattleMessenger**.

Enums: A classe Enums é um grande banco de dados que centraliza todos os enums do jogo, permitindo uma fácil referência a tipos, tiers, elementos, tipos de dano, fases de batalha, e status. Por ser uma classe global, ela está sempre instanciada e acessível em todo o código do jogo. Cada enum fornece um conjunto de valores pré-definidos que são usados para categorizar e controlar diferentes aspectos dos Digimons, habilidades, e fases da batalha.

**Enums principais:**

1. **Type** (Tipos de Digimon):
   * **VIRUS, DATA, VACCINE, FREE, SPECIAL**: Tipos que categorizam os Digimons com base em sua natureza e atributos. Esses tipos podem afetar interações em batalha.
2. **Tier** (Tiers de Digimon):
   * **ROOKIE, CHAMPION, ULTIMATE, MEGA**: Representa o nível de poder dos Digimons, desde os mais básicos (Rookie) até os mais poderosos (Mega).
3. **Element** (Elementos):
   * **NEUTRAL, FIRE, WATER, PLANT, THUNDER, EARTH, WIND, LIGHT, DARK**: Elementos associados aos Digimons e habilidades, determinando vantagens e desvantagens em combate.
4. **DamageType** (Tipos de Dano):
   * **PHYSICAL, MAGICAL, TRUE**: Tipos de dano que afetam como o dano é calculado e quais resistências ou defesas podem se aplicar.
5. **DamageSubType** (Subtipos de Dano):
   * **DIRECT, OVERTIME, DEFLECT, SPECIAL**: Subtipos de dano que definem como o dano é aplicado ao longo do tempo ou em circunstâncias especiais.
6. **BattlePhase** (Fases da Batalha):
   * **BATTLESTART, TURNSTART, CHOICE, ACTION, POSACTION, TURNEND, BATTLEEND**: Fases que controlam o fluxo da batalha, desde o início até o fim.
7. **StatusType** (Tipos de Efeito):
   * **BUFF, DEBUFF**: Define se o efeito aplicado ao Digimon é um buff (positivo) ou debuff (negativo).

HUDDisplay: A classe HUDDisplay é responsável por montar e atualizar os elementos visuais do HUD do Digimon, exibindo informações como nome, nível, vida e mana. Esses elementos são apresentados ao jogador e atualizados conforme o status do Digimon muda durante a batalha. A classe utiliza elementos como RichTextLabel para mostrar o nome e nível do Digimon, e ProgressBar para mostrar a barra de vida e mana.

**Propriedades:**

1. **Elementos em cena**:
   * **nameAndLevel: RichTextLabel**: Exibe o nome e nível do Digimon de forma centralizada.
   * **health: ProgressBar**: Barra de progresso que representa a saúde (vida) do Digimon.
   * **mana: ProgressBar**: Barra de progresso que representa a mana do Digimon.
   * **ownerPlayer: Tamer**: O tamer que controla o Digimon, referenciando o dono do HUD.
   * **relatedDigimon: Digimon**: O Digimon associado ao **Tamer** e ao HUD.

**Métodos principais:**

1. **\_ready() -> void**:
   * Inicializa a referência para o Digimon associado ao tamer.
   * Ajusta o modo de preenchimento da barra de saúde para o inimigo, fazendo com que ela preencha da direita para a esquerda se o tamer for o inimigo.
2. **setHealth() -> void**:
   * Define o valor máximo e o valor atual da barra de saúde, baseado na vida atual e máxima do Digimon.
   * Utiliza a função **Util.cap()** para garantir que o valor não exceda 100%.
3. **setMana() -> void**:
   * Define o valor máximo e o valor atual da barra de mana, baseado na mana atual e máxima do Digimon.
   * Também utiliza **Util.cap()** para garantir que o valor esteja dentro do limite permitido.
4. **updateValues() -> void**:
   * Atualiza o texto exibido no **nameAndLevel**, mostrando o nome e nível atual do Digimon.
   * Chama os métodos **setHealth()** e **setMana()** para atualizar as barras de vida e mana com base nos valores mais recentes do Digimon.

InfoWindow: A classe InfoWindow é responsável por exibir uma janela de informações detalhadas quando o jogador mantém pressionado um botão no painel de habilidades ou ações. Essa janela pode mostrar o nome e a descrição da habilidade ou item relacionado ao botão pressionado. A classe também interage com o ButtonPanel, bloqueando os outros botões enquanto a janela está visível para garantir que o jogador esteja focado nas informações exibidas.

**Propriedades:**

1. **Elementos em cena**:
   * **buttonPanel: ButtonPanel**: Referência ao painel de botões, para bloquear e desbloquear os botões quando a janela de informação é mostrada ou escondida.
   * **title: RichTextLabel**: Exibe o título (geralmente o nome da habilidade ou item).
   * **description: RichTextLabel**: Exibe a descrição detalhada da habilidade ou item.

**Métodos principais:**

1. **\_ready() -> void**:
   * Inicializa a janela de informações como invisível. Ela só será visível quando o jogador pressionar um botão e mantiver pressionado.
2. **updateDisplay(ntitle: String, ncontent: String) -> void**:
   * Atualiza o título e a descrição da janela de informação com base nos parâmetros fornecidos.
   * Os textos são formatados com tags de centralização para uma apresentação mais clara e organizada.
3. **showWindow(ntitle: String, ncontent: String) -> void**:
   * Mostra a janela de informações, bloqueando os outros botões no **ButtonPanel** para evitar interações enquanto a janela está visível.
   * Atualiza o conteúdo da janela e torna-a visível na tela.
   * Define o valor descWindowOn do **ButtonPanel** como true para indicar que a janela de descrição está aberta.
4. **hideWindow() -> void**:
   * Esconde a janela de informações e desbloqueia os botões no **ButtonPanel**.
   * Define descWindowOn como false para indicar que a janela de descrição foi fechada.

NumberSpawner: Apesar do nome, a classe NumberSpawner não exibe apenas números. Ela é responsável por mostrar na tela qualquer tipo de informação pontual durante a batalha, incluindo danos recebidos, status aplicados, ou outras mensagens relevantes. Essa classe usa diferentes cores e formatações para exibir as informações, diferenciando-as por tipo de conteúdo, como dano de elementos, status buffs/debuffs, ou mensagens de texto.

**Propriedades:**

1. **Cena estática**:
   * **spawn: PackedScene**: A cena pré-carregada que será instanciada para exibir as informações na tela. Essa cena é o componente que realmente mostra o conteúdo ao jogador.
2. **Dicionários de cores**:
   * **elementColor: Dictionary**: Dicionário que associa cada elemento a uma cor específica. As cores são usadas para exibir o dano com base no elemento associado.
   * **statusColor: Dictionary**: Dicionário que associa buffs e debuffs a cores específicas, permitindo a exibição diferenciada de status aplicados.

**Métodos principais:**

1. **spawnContent(content) -> void**:
   * Responsável por instanciar um novo objeto de exibição na tela e definir o conteúdo a ser mostrado.
   * Identifica o tipo de conteúdo (dano, status ou string) e chama a função apropriada para processar esse conteúdo.
   * Utiliza call\_deferred("add\_child", newSpawn) para adicionar o novo elemento à cena de forma assíncrona.
2. **processDamageData(damageData: DamageData) -> String**:
   * Processa informações de dano, determinando a cor com base no elemento do dano e formatando a exibição de dano crítico ou normal.
   * O dano é formatado com um contorno colorido e uma fonte de tamanho maior se for um dano crítico.
3. **processStatus(statusEffect: StatusEffect) -> String**:
   * Processa a exibição de status aplicados (buffs e debuffs), usando cores específicas para distinguir entre buffs e debuffs.
   * Exibe o nome do status aplicado com formatação centralizada e contorno colorido.
4. **processString(string: String) -> String**:
   * Processa mensagens de texto gerais e formata-as para exibição centralizada com uma cor padrão (usada para informações pontuais que não estão associadas a dano ou status).

Skill: A classe Skill é a classe base para todas as habilidades (skills) que podem ser usadas por um Digimon. Ela define os atributos e comportamentos fundamentais de uma habilidade, como nome, descrição, custo de mana, prioridade, e cooldown. Como a classe é abstrata, ela nunca é instanciada diretamente, mas serve de base para classes filhas que implementam habilidades específicas.

**Propriedades:**

1. **Identificação**:
   * **skillId: int**: Identificador único da habilidade.
   * **skillName: String**: Nome da habilidade.
   * **skillDescritption: String**: Descrição da habilidade.
   * **skillIcon: CompressedTexture2D**: Ícone da habilidade usado na interface.
   * **manaCost: int**: Custo de mana necessário para usar a habilidade.
2. **Elementos de controle**:
   * **staticPriority: int**: Prioridade estática da habilidade, definida em sua criação.
   * **priority: int**: Prioridade dinâmica, calculada em tempo real durante a batalha.
   * **usable: bool**: Indica se a habilidade está disponível para uso (pode ser desativada por cooldown).
   * **accuracy: int**: Precisão da habilidade.
   * **cooldowm: int**: Tempo de recarga (cooldown) da habilidade após o uso.
   * **currentCooldown: int**: Tempo restante até que a habilidade possa ser usada novamente (inicialmente zero).
3. **Elementos de cena**:
   * **texture: CompressedTexture2D**: Textura visual associada à habilidade.
   * **textureRange: int**: Número de frames necessário para a aplicação da textura.

**Métodos principais:**

1. **learn(learner: Digimon, index: int) -> bool**:
   * Associa a habilidade a um **Digimon** no índice especificado, permitindo que o Digimon aprenda essa habilidade.
   * Retorna true indicando que a habilidade foi aprendida com sucesso.
2. **unlearn(\_unlearner: Digimon) -> void**:
   * Método abstrato, que será implementado em classes filhas, para "desaprender" ou remover uma habilidade de um **Digimon**.
3. **applyCooldown() -> void**:
   * Aplica o cooldown à habilidade após o uso, definindo o valor de **currentCooldown** e desativando a habilidade (tornando-a não **usable**).
4. **countCooldown() -> void**:
   * Reduz o cooldown da habilidade no final de cada turno. Quando o cooldown chega a zero, a habilidade é reativada (marcada como **usable**).
5. **effect(\_digimon: Digimon) -> void**:
   * Método abstrato que será implementado nas classes filhas para definir o efeito da habilidade sobre o Digimon alvo.
6. **priorityCheck(\_digimon: Digimon) -> void**:
   * Método abstrato para verificar a prioridade da habilidade. Essa função é chamada antes de calcular a prioridade da habilidade.
7. **priorityCalculation(digimon: Digimon) -> void**:
   * Calcula a prioridade da habilidade no contexto da batalha. Se a habilidade não for usável, estiver em cooldown, ou o Digimon não tiver mana suficiente, a prioridade será definida como 0.

SkillSpawner: A classe SkillSpawner é responsável por projetar os efeitos visuais das habilidades durante a batalha. Ela recebe uma habilidade, exibe seu efeito visual correspondente (usando sprites e animações), e depois retorna o controle para o Digimon para que o efeito da habilidade seja processado. Funciona como um intermediário que garante que o jogador ou o oponente veja o efeito visual de cada habilidade antes de a habilidade ser aplicada.

**Propriedades:**

1. **Habilidade recebida**:
   * **currentSkill: Skill**: A habilidade atual que o **SkillSpawner** está exibindo. A habilidade é atribuída quando o método **spawSkill()** é chamado.
2. **Elementos de cena**:
   * **relatedDigimon: Digimon**: O Digimon relacionado a essa instância do **SkillSpawner**, que usará a habilidade.
   * **skillSprite: Sprite2D**: O sprite usado para exibir a textura visual da habilidade.
   * **animator: AnimationPlayer**: Controla a animação do sprite durante a exibição da habilidade.

**Métodos principais:**

1. **\_ready() -> void**:
   * Inicializa a cena, tornando o **SkillSpawner** invisível por padrão.
   * Se o **relatedDigimon** for controlado pelo jogador (**Player**), inverte a direção do sprite para garantir que os efeitos visuais fiquem voltados corretamente para o oponente.
2. **spawSkill(skill: Skill) -> void**:
   * Executa o efeito visual da habilidade.
   * Atribui a habilidade atual ao **currentSkill** e define a textura do **skillSprite** com base na habilidade.
   * Define o número de frames do sprite de acordo com o **textureRange** da habilidade.
   * Toca a animação correspondente à habilidade, e torna o **SkillSpawner** visível.
   * Notifica o **BattleManager** que uma ação está em execução com **relatedDigimon.BTM.inAction()**.
3. **\_onAnimationFinished(anim\_name: String) -> void**:
   * Chamado quando a animação do efeito visual da habilidade é concluída.
   * Torna o **SkillSpawner** invisível novamente.
   * Se o nome da animação tiver "action", significa que a habilidade foi exibida, então o efeito da habilidade é processado chamando **relatedDigimon.gotTargeted(currentSkill)**.
   * Notifica o **BattleManager** que a ação foi concluída com **relatedDigimon.BTM.outAction("Skill Spawner")**.

SpawnedNumber: A classe SpawnedNumber é uma célula visual que, quando instanciada na cena, exibe o conteúdo recebido (como números de dano, informações de status, ou qualquer outro tipo de mensagem). Após exibir o conteúdo, ela realiza uma animação simples: o conteúdo sobe na tela enquanto desbota (fade out) até desaparecer completamente. Quando o fade out termina, a célula se autodestrói, removendo-se da cena. Essa classe é usada para dar feedback visual instantâneo e temporário durante a batalha.

**Propriedades:**

1. **Elementos de cena**:
   * **currentNumber: RichTextLabel**: O texto ou número a ser exibido.
   * **outTimer: Timer**: Um temporizador que controla o tempo restante até que o objeto se autodestrua.
2. **Variáveis de controle de animação**:
   * **limitPosition: int**: A altura máxima que o número pode alcançar enquanto sobe.
   * **currentPosition: int**: A posição atual do número enquanto sobe.
   * **velocity: int**: A velocidade com que o número se move verticalmente.
   * **moduleRatio: float**: A taxa de desvanecimento (fade out) da opacidade da célula.

**Métodos principais:**

1. **\_process(\_delta) -> void**:
   * Este método é chamado a cada quadro (frame) para mover a célula verticalmente e reduzir gradualmente sua opacidade.
   * A posição Y é atualizada conforme o número sobe, até atingir o valor de **limitPosition**.
   * A opacidade é reduzida de acordo com o valor de **moduleRatio**, até que chegue a zero, momento em que o temporizador **outTimer** é iniciado.
   * Quando a opacidade chega a zero, o processo de animação é interrompido com **set\_process(false)**.
2. **showNumber(number: String) -> void**:
   * Exibe o conteúdo fornecido (número ou texto) e reinicia o processo de animação.
   * A opacidade do objeto é restaurada para 1 (totalmente visível), e o texto é atualizado com o valor de **number**.
3. **outTimeOut() -> void**:
   * Método chamado quando o temporizador **outTimer** expira.
   * Remove a célula da cena com **queue\_free()**, destruindo o objeto após a exibição completa de sua animação.

Translator: A classe Translator é usada para gerenciar a tradução da aplicação, permitindo a alternância entre diferentes idiomas e o carregamento dinâmico de dicionários de tradução. Ela utiliza arquivos de recurso (.tres) e arquivos de configuração JSON para armazenar e atualizar as mensagens de tradução. A classe também lida com o ajuste da língua atual e garante que as traduções sejam salvas e carregadas corretamente.

**Propriedades:**

1. **currentMessageSize: int**:
   * O número de mensagens que devem estar no dicionário de tradução. Isso garante que o dicionário seja completo e atualizado.
2. **languages: Dictionary**:
   * Um dicionário que mapeia os idiomas disponíveis para os arquivos .tres que contêm as traduções. As chaves são os códigos de idioma ("en" para inglês e "pt" para português).
3. **configLogPath: Dictionary**:
   * Um dicionário que mapeia os idiomas para seus respectivos arquivos JSON de configuração. Esses arquivos são usados para carregar mensagens adicionais de tradução.
4. **currentLanguage: String**:
   * O idioma atualmente em uso na aplicação.
5. **currentDictionary: Translation**:
   * O dicionário de tradução atual, carregado a partir do arquivo .tres correspondente ao idioma selecionado.

**Métodos principais:**

1. **\_ready() -> void**:
   * Método chamado quando a classe é inicializada. Se a variável currentLanguage não estiver definida, define o idioma padrão como inglês e chama changeLanguage("en").
2. **changeLanguage(newLanguage: String) -> void**:
   * Muda o idioma da aplicação para o idioma especificado por newLanguage.
   * Carrega o dicionário de tradução do idioma selecionado e verifica se o número de mensagens no dicionário está correto. Se o dicionário estiver incompleto, chama o método **translationConfig()** para atualizar as mensagens de tradução.
3. **translationConfig() -> void**:
   * Carrega o arquivo JSON de configuração do idioma atual e verifica se há novas mensagens de tradução para serem adicionadas ao dicionário.
   * Se houver mensagens no JSON que ainda não estão no dicionário, elas são adicionadas e o dicionário é atualizado.
4. **saveTranslation() -> void**:
   * Salva o dicionário de tradução atualizado permanentemente, sobrescrevendo o arquivo .tres correspondente ao idioma atual.
5. **setTranslator() -> void**:
   * Define o dicionário de tradução atual no **TranslationServer**, que gerencia as traduções no Godot.
   * Define o idioma atual da aplicação no **TranslationServer**.

Utility: A classe Utility oferece várias funções utilitárias para manipular tipos, elementos, probabilidades, e cálculos numéricos, além de gerar valores aleatórios. Ela é essencial para realizar operações comuns de maneira eficiente e modular em todo o jogo. A classe lida com conversões de tipos, ajustes de proporções, criação de dados de dano e cálculo de relações de tipos e elementos.

**Propriedades:**

1. **Dicionários de conversão**:
   * **convertType: Dictionary**: Mapeia os tipos de Digimon (como VIRUS, DATA, etc.) para valores numéricos que podem ser usados nos cálculos.
   * **convertElement: Dictionary**: Mapeia os elementos (como FIRE, WATER, etc.) para valores numéricos que podem ser usados para calcular relações elementares.
2. **Dicionários de relação**:
   * **typeRatio: Dictionary**: Define as relações de vantagem e desvantagem entre diferentes tipos de Digimons (como VIRUS vs. VACCINE).
   * **elementRatio: Dictionary**: Define as relações de vantagem e desvantagem entre diferentes elementos (como FIRE vs. WATER).

**Métodos principais:**

1. **random(minn: int, maxn: int) -> int**:
   * Gera um número aleatório entre os valores mínimos e máximos especificados.
2. **chance(nchance: float) -> bool**:
   * Retorna true ou false com base em uma chance percentual. Se a chance for 100%, retorna sempre true. Caso contrário, gera um número aleatório e verifica se está dentro do limite especificado pela chance.
3. **cap(value: float) -> float**:
   * Limita um número flutuante (float) a duas casas decimais. Útil para manter os valores numéricos do jogo precisos e legíveis.
4. **damageDataBuilder(damageSkill: DamageSkill) -> DamageData**:
   * Cria um novo objeto **DamageData** com base nas informações da habilidade de dano (**DamageSkill**). Ele coleta informações sobre o tipo de dano, elemento, atacante, e valor do dano.
5. **getProportion(portion: float, whole: float) -> float**:
   * Calcula a proporção entre dois valores (por exemplo, entre a vida atual e a vida total do Digimon).
6. **getElementRatio(atacker: Enums.Element, defender: Enums.Element) -> float**:
   * Calcula a vantagem ou desvantagem entre dois elementos. Retorna um multiplicador que indica se o atacante tem vantagem (1.3), desvantagem (0.7) ou é neutro (1.0) em relação ao defensor.
7. **getTypeRatio(atacker: Enums.Type, defender: Enums.Type) -> float**:
   * Calcula a relação de tipo entre o atacante e o defensor, usando o dicionário **typeRatio**. Retorna um multiplicador indicando a vantagem ou desvantagem entre tipos.
8. **pickOne(options: Array) -> Object**:
   * Seleciona aleatoriamente um item de uma lista (array) de opções e retorna o item selecionado.