Jogo da Cobrinha (RUST-GGEZ)

Welder Carlos Siqueira

Conceitos Iniciais para implementação (Jogo Cobrinha)

Importações de Bibliotecas

As bibliotecas ggez, rand e outras são importadas no início do código para fornecer funcionalidades necessárias para o jogo.

importar as bibliotecas necessárias para o jogo, incluindo ggez, rand, std::collections, e std::time.

```
// Primeiro, importaremos as crates necessárias para o nosso jogo;
// neste caso, apenas `ggez` e `rand`.
// Em seguida, precisamos realmente "usar" as partes do ggez que iremos
// precisar com frequência.
use ggez::event::KeyCode;
use ggez::{event, graphics, Context, GameResult};

// Vamos trazer algumas coisas do `std` para nos ajudar no futuro.
use std::collections::LinkedList;
use std::time::{Duration, Instant};

// E, finalmente, trazemos a trait `Rng` para o escopo para que possamos gerar
// alguns números aleatórios mais tarde.
use rand::Rng;
```

Definição de Constantes(Jogo da Cobrinha)

```
// Aulas iniciais definição de constantes(Constrrutores, Métodos, Traits, etc.)
//Define várias constantes, incluindo o tamanho da grade do jogo, tamanho das células da grade,
// tamanho da janela, taxa de atualização tentativa para  milissegundos por atualização.
const GRID_SIZE: (i16, i16) = (30, 20);
const GRID_CELL_SIZE: (i16, i16) = (32, 32);
const SCREEN_SIZE: (u32, u32) = (GRID_SIZE.0 as u32 * GRID_CELL_SIZE.0 as u32, GRID_SIZE.1 as u32 * GRID_CELL_SIZE.1 as u32);
const UPDATES_PER_SECOND: f32 = 8.0;
const MILLIS_PER_UPDATE: u64 = (1.0 / UPDATES_PER_SECOND * 1000.0) as u64;
```

As constantes como GRID_SIZE e UPDATES_PER_SECOND são definidas para configurar o tamanho do tabuleiro, a taxa de atualização e outras configurações importantes do jogo.

Struct GridPosition

Essa struct representa uma posição na grade do jogo, com coordenadas x e y. Implementa a trait ModuloAssinado para realizar operações de módulo em números negativos.

Fornece métodos para criar novas posições e calcular posições após um movimento.

```rust
// derive é usado para derivar automaticamente as traits Clone, Copy, PartialEq, Eq e Debug para a struct GridPosition.
//a escolha de x:i16 e y:i16 é para que a posição possa ser representada por números negativos.
#[derive(Clone, Copy, PartialEq, Eq, Debug)]
6 implementations
struct GridPosition {
    x: i16,
    y: i16,
}
// fn modulo(&self, n: Self) -> Self; é usado para implementar o módulo assinado para a struct GridPosition.
//voltar e tentar refazer 03/09/2023
1 implementation
trait ModuloAssinado {
    fn modulo(&self, n: Self) -> Self;
}
//impl mostra que a struct GridPosition implementa a trait ModuloAssinado.
// atrait ModuloAssinado é implementada para operações de módulo com números negativos.
//ops é usado para implementar operadores.std::ops::Add é usado para implementar a adição.
// finalmente self
impl<T> ModuloAssinado for T
where
    T: std::ops::Add<Output = T> + std::ops::Rem<Output = T> + Clone,
{
    fn modulo(&self, n: T) -> T {
        (self.clone() % n.clone() + n.clone()) % n
    }
}
 // Fornece métodos auxiliares para criação de posições e movimentos
//Define a estrutura GridPosition para representar a posição na grade do jogo.
//Implementa a trait ModuloAssinado para operações de módulo com números negativos.
//thread_rng é usado para gerar números aleatórios.
impl GridPosition {
    pub fn new(x: i16, y: i16) -> Self {
        GridPosition { x, y }
    }

    pub fn aleatoria(max_x: i16, max_y: i16) -> Self {
        let mut rng: ThreadRng = rand::thread_rng();
        (rng.gen_range(0..max_x), rng.gen_range(0..max_y)).into()
    }
```

Enum Direção

Uma enumeração que representa as direções possíveis que a cobra pode se mover.

Possui métodos para obter a direção inversa e converter um KeyCode (tecla pressionada) em uma direção.

```rust
//O KEY CODE é usado para representar os códigos das teclas do teclado.
//aqui recebe um KeyCode e retorna uma Option<Direcao> que pode ser Some(Direcao) ou None.
// clone é usado para que o KeyCode não seja movido.
//copy é usado para que o KeyCode seja copiado.
//partialeq é usado para que o KeyCode possa ser comparado com outros KeyCodes.
//eq é usado para que o KeyCode possa ser comparado com outros KeyCodes.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
6 implementations
enum Direcao {
    Cima,
    Baixo,
    Esquerda,
    Direita,
}
// de inicio a cobra começa indo para a direita.
impl Direcao {
    pub fn inversa(&self) -> Self {
        match *self {
            Direcao::Cima => Direcao::Baixo,
            Direcao::Baixo => Direcao::Cima,
            Direcao::Esquerda => Direcao::Direita,
            Direcao::Direita => Direcao::Esquerda,
        }
    }
//as direções possíveis da cobra.
    pub fn de_keycode(keycode: KeyCode) -> Option<Direcao> {
        match keycode {
            KeyCode::Cima => Some(Direcao::Cima),
            KeyCode::Baixo => Some(Direcao::Baixo),
            KeyCode::Esquerda => Some(Direcao::Esquerda),
            KeyCode::Direita => Some(Direcao::Direita),
            _ => None,
        }
    }
}
```

Segmento da Cobra (Partes)

Define a estrutura Segmento para representar um segmento da cobra.

```rust
//Segmento da cobra aqui a ideia Define a estrutura Segmento para representar um segmento da cobra.
#[derive(Clone, Copy, Debug)]
4 implementations
struct Segmento {
    //Gridposition faz parte da biblioteca ggez e é usado para representar a posição do segmento da cobra na grade.
    pos: GridPosition,
}
// fazendo a cobra se mover
impl Segmento {
    pub fn novo(pos: GridPosition) -> Self {
        Segmento { pos }
    }
}
```

Impl Comida

Define a estrutura Comida para representar a comida que a cobra pode comer.

Implementa um método desenhar para desenhar a comida na tela.

```
//nesta função a ideia é desenhar um segmento da cobra na tela.
1 implementation
struct Comida {

    pos: GridPosition,
}
//Define a estrutura Comida para representar a comida que a cobra pode comer.
impl Comida {
    // para que a comida possa ser criada em qualquer posição da grade.
    pub fn nova(pos: GridPosition) -> Self {
        Comida { pos }
    }
//cor após comer ajuda a gerar uma nova comida aleatória quando a cobra come a comida.
    fn desenhar(&self, ctx: &mut Context) -> GameResult {
        //let mesh funciona como um ponteiro para a comida.
        let mesh: ! = graphics::MeshBuilder::new()
            .retângulo(
                //aqui a ideia é criar um retângulo com o tamanho de uma célula da grade.
                graphics::DrawMode::fill(),
                // drawmode é usado para definir o modo de desenho do retângulo.
                self.pos.into(),
                graphics::Color::new(0.0, 0.0, 1.0, 1.0),
            )?
            .construir(ctx)?;
//por fim, desenha a comida na tela.
//graphics faz parte da biblioteca ggez e é usado para desenhar a comida na tela.
        graphics::draw(ctx, &mesh, graphics::DrawParam::default())?;
        Ok(())
    }
}
```

Enum Comeu

Enums contém as opções de estado após comer ou se comeu ou se comeu a si mesma assim , dando fim ao jogo.

```rust
//vendo as duas opç~es posivés de comeu a comida ou a si mesma
#[derive(Clone, Copy, Debug)]
3 implementations
enum Comeu {
    SiMesma,
    Comida,
}

// aqui como a ira se comportar a cobra
//a struct  a cobra é composta por uma cabeça, uma direção, um corpo, uma opção de Comeu e a última direção de atualização.
1 implementation
struct Cobra {
    cabeça: Segmento,
    dir: Direcao,
    corpo: LinkedList<Segmento>,
    comeu: Option<Comeu>,
    última_direção_atualização: Direcao,
}
//o impl é usado para implementar métodos para a struct Cobra.
// Observação Não use ENUMS para representar a direção da cobra, pois isso dificulta a implementação de alguns métodos.(Só da ERRO)
//O LINKEDLIST é usado para representar o corpo da cobra, pois é uma estrutura de dados que permite inserção e remoção de elementos
//em qualquer posição.
//O ELTOPTION é usado para representar a opção de Comeu, pois é uma estrutura de dados que
//permite representar um valor ou a ausência de um valor.
//COBRA SEGMENTO novo é usado para criar uma nova cobra com uma posição inicial.
impl Cobra {
    pub fn nova(pos: GridPosition) -> Self {
        let mut corpo: LinkedList<Segmento> = LinkedList::new();
        corpo.push_back(elt: Segmento::novo(pos: (pos.x - 1, pos.y).into()));
        Cobra {
            cabeça: Segmento::novo(pos),
            dir: Direcao::Direita,
            última_direção_atualização: Direcao::Direita,
            corpo,
            comeu: None,
        }
    }
```

Função Atualizar

Lida com a lógica de atualização do jogo, incluindo movimento da cobra, verificação de colisões e atualização da posição da comida.

E também a movimentação da cobra e verificação de colisões

Define a estrutura Estado Jogo que implementa o EventHandler do ggez para controlar o estado do jogo.

```rust
//agora aqui na fnção comeu a ideia é verificar se a cobra comeu a comida.
//e self.cabeça.pos == comida.pos verifica se a cabeça da cobra está na mesma posição da comida.(IMPORTANTE)
//ajuda Leonardo 04/09/2023
    fn comeu(&self, comida: &Comida) -> bool {
        self.cabeça.pos == comida.pos
    }
//caso ela tenha comido a si mesma de inicio verifica com uso do bool se a cobra comeu a si mesma.
// no laço for seg in self.corpo.iter() verifica se a cabeça da cobra está na mesma posição de algum segmento do corpo da cobra.
// e assim o iter() é usado para iterar sobre os elementos da lista do corpo da cobra.
    fn comeu_si_mesma(&self) -> bool {
        for seg: &Segmento in self.corpo.iter() {
            if self.cabeça.pos == seg.pos {
                return true;
            }
        }
        false
    }
// na função atualizar a ideia é atualizar a cobra após um intervalo de tempo.
// em if.self.dir == self.última_direção_atualização verifica se a cobra está tentando ir na direção oposta à sua última direção de atualizaçã
//self corpo.is_none() verifica se a cobra comeu alguma coisa.
// e por fim self.ultimo_direcao_atualizacao = self.dir atualiza a última direção de atualização da cobra.
    fn atualizar(&mut self, comida: &Comida) {
        let nova_pos_cabeça: GridPosition = GridPosition::nova_apos_movimento(self.cabeça.pos, self.dir);
        let nova_cabeça: Segmento = Segmento::novo(pos: nova_pos_cabeça);
        self.corpo.push_front(elt: self.cabeça);
        self.cabeça = nova_cabeça;
        if self.comeu_si_mesma() {
            self.comeu = Some(Comeu::SiMesma);
        } else if self.comeu(comida) {
            self.comeu = Some(Comeu::Comida);
        } else {
            self.comeu = None;
        }
        if self.comeu.is_none() {
            self.corpo.pop_back();
        }
        self.última_direção_atualização = self.dir;
    }
```

Função drawing(Desenho)

Define a estrutura Cobra para representar a cobra no jogo.

Implementa métodos update e draw para atualização e desenho do jogo.

```
//para a função desenhar a ideia é desenhar a cobra na tela.
// em (&self, ctx: &mut Context) recebe uma referência imutável para a cobra e uma referência mutável para o contexto.
// o mutavel para o contexto é necessário para desenhar a cobra na tela.
//for seg in self.corpo.iter() itera sobre os segmentos do corpo da cobra.
//em graphics:: draw(ctx, &mesh, graphics::DrawParam::default()) desenha o segmento da cobra na tela.
// let mesh = graphics::MeshBuilder::new() cria um novo mesh para desenhar o segmento da cobra.
// em .rectangle() cria um retângulo com o tamanho de uma célula da grade.
    fn desenhar(&self, ctx: &mut Context) -> GameResult {
        for seg: &Segmento in self.corpo.iter() {
            let mesh: ! = graphics::MeshBuilder::new()
                .retângulo(
                    graphics::DrawMode::fill(),
                    seg.pos.into(),
                    graphics::Color::new(0.5, 0.0, 0.0, 1.0),
                )?
                .construir(ctx)?;
            graphics::draw(ctx, &mesh, graphics::DrawParam::default())?;
        }
        let mesh: ! = graphics::MeshBuilder::new()
            .retângulo(
                graphics::DrawMode::fill(),
                self.cabeça.pos.into(),
                graphics::Color::new(1.0, 0.0, 0.0, 1.0),
            )?
            .construir(ctx)?;
        graphics::draw(ctx, &mesh, graphics::DrawParam::default())?;
        Ok(())
    }
} impl Cobra
```

Função Main

```
▶ Run | Debug
fn main() -> GameResult {
    // Primeiro, criamos uma estrutura `ggez::Conf` que define as configurações do nosso jogo.
    let (ctx: Context, event_loop: EventLoop<()>) = ggez::ContextBuilder::new(game_id: "Jogo da Cobrinha-=TECVII", author: "welder") ContextBuil
        // Primeiro, criamos uma estrutura `ggez::Conf` que define as configurações do nosso jogo.
        .window_setup(ggez::conf::WindowSetup::default().title("Jogo da Cobrinha!!!")) ContextBuilder
       // Em seguida, chamamos a função `ggez::ContextBuilder::new` para criar um `ContextBuilder`,
    // que nos permitirá personalizar como queremos que o contexto seja criado.
        .window_mode(
            ggez::conf::WindowMode::default()
                .dimensions(SCREEN_SIZE.0 as f32, SCREEN_SIZE.1 as f32),
        ) ContextBuilder

        // CASO DE ERRADO, VAI DAR RUIM!!!!!!
        .build()
        .expect("DEU RUIM!!!!!!");
   // Em seguida, criamos uma instância do nosso `GameState` e a passamos para a função
    // `ggez::event::run` para iniciar o loop principal do jogo.
    let state = GameState::new()?;

    event::run(ctx, event_loop, state)
}
```

Por fim a função Main que configura o contexto do jogo e inicia o loop principal do ggez para executar o jogo.

OBRIGADO!!!!!