Rust+Fyrox游戏开发大作业

本次项目是基于Rust和Fyrox2D游戏基础以及相关素材进行的Rust程序实践。

这个项目是一款名为《天空城勇士》的游戏，玩家通过操控具体的角色进行关卡的游玩、怪物的击打以及人物的探险。由于人力和时间有限，便使用了Fyrox教程中的基本素材和技术指导，比如关卡障碍物、人物角色的设计、怪物骷髅的贴图设计等等，但代码的功能具体实现由本人独立完成，协助于ChatGPT

《天空城勇士》项目的构建设计一共分为三个板块。

- 通过fyrox的editor进行游戏关卡场景的基本创建，例如角色站立平台的创建，背景树木以及箱子等等障碍物的摆放。
- 添加游戏角色，实现游戏人物角色的具体贴图，以及基本动作和场景的交互
- 添加游戏怪物，实现怪物的具体贴图，基本移动原理以及和角色人物的交互

PART1 基于Fyrox图形编辑器的游戏场景构建

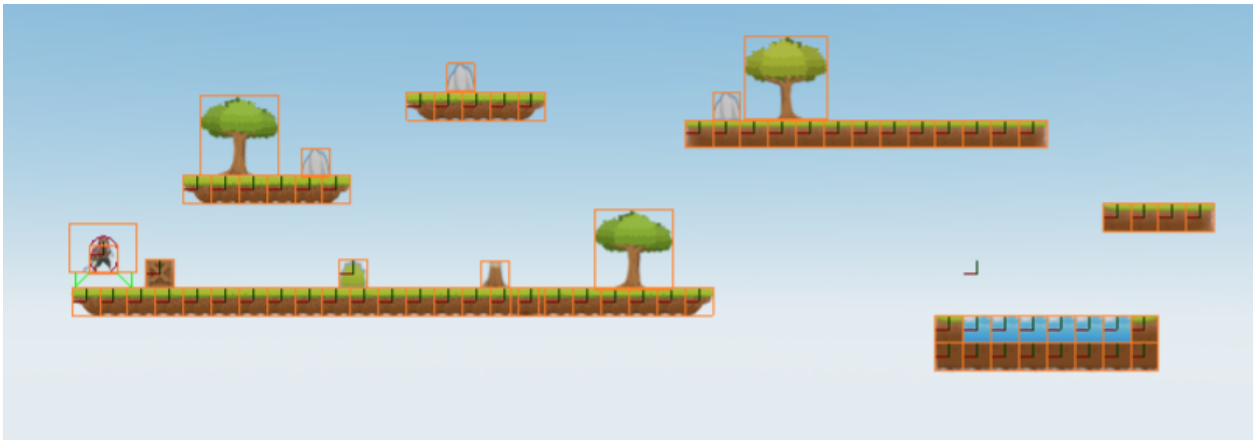
在项目文件夹下，使用命令行打开图形编辑器。

```
cargo run --package editor --release
```

使用命令直接运行游戏。

```
cargo run --package executor --release
```

构建如图所示场景。



场景的具体构建操作流程：

在editor中的ROOT下面添加刚体，并为刚体添加碰撞体和精灵，分别实现物理碰撞和图形贴图，以及动画实现。在场景的构建当中并未使用动画。按照需要构建空岛道路、水池、树木、石头、箱子、灌木等等。

PART 2 构建人物角色并实现人物的基本移动和操作以及动画

同样的，构建一个刚体，人物采用的是胶囊状的碰撞体，更符合角色人物的物理碰撞，以及添加精灵。并为精灵添加基本的贴图。代码实现：在lib.rs中创建结构体Player并添加需要的属性。

- 实现人物的左右移动。通过按键A、D、W和S来操纵人物左右移动和跳跃下蹲，具体代码实现：
  - 在lib.rs中，通过on_os_event来检测键盘的输入：

```
fn on_os_event(&mut self, event: &Event<()>, context: &mut
        if let Event::WindowEvent { event, .. } = event {
            if let WindowEvent::KeyboardInput { event, ..
                if let PhysicalKey::Code(keycode) = event.
                    let is_pressed = event.state == Elemer
                    match keycode {
                        //设置结构体中的按键字段并输出按键信息
                        KeyCode::KeyA => {
```

```
                            self.move_left = is_pressed;
                            println!("Move left: {}", is_p
                        }
                        KeyCode::KeyD => {
                            self.move_right = is_pressed;
                            println!("Move right: {}", is_
                        }
                        KeyCode::KeyW => {
                            self.jump = is_pressed;
                            println!("Jump: {}", is_presse
                        }
                        KeyCode::KeyS => {
                            self.ddown = is_pressed;
                            println!("down: {}", is_presse
                        }
                        ......
```

- 在on_update函数中，通过按键信息传达的属性值move_left和move_right来改变x_speed值，改变刚体的运动速度，实现人物的移动。同时由于是2D的游戏，在改变行进方向时需要将人物进行反转。代码具体实现

```
fn on_update(&mut self, context: &mut ScriptContext) {
        if let Some(rigid_body) = context.scene.graph[cont
            // 处理移动速度
            let x_speed = if self.move_left {
                1.5
            } else if self.move_right {
                -1.5
            } else {
                0.0
            };
            // 处理跳跃
            rigid_body.set_lin_vel(Vector2::new(x_speed, r
            if self.jump {
                rigid_body.set_lin_vel(Vector2::new(x_spee
```

```
            }
            if self.ddown {
                self.current_animation = 5;
            }
            // 处理角色方向
            if let Some(sprite) = context.scene.graph.try_
                if x_speed != 0.0 {
                    let local_transform = sprite.local_tra
                    let current_scale = **local_transform.
                    local_transform.set_scale(Vector3::new
                        current_scale.x.copysign(-x_speed),
                        current_scale.y,
                        current_scale.z,
                    ));
                }
            }
            .....
```

- 同时在角色移动的时候添加动画，使之更具有移动感。代码实现：

  在角色的结构体中添加动画的Vec数组，存放多个不同的动画。并通过索引在editor里面为角色添加动画片段，使之连续播放成为动画。

```
//角色属性
struct Player {
    sprite: Handle<Node>,
    move_left: bool,
    move_right: bool,
    jump: bool,
    attack1: bool,
    attack2: bool,
    attack3: bool,
    ddown: bool,
    health: i32,
    //添加动画
    animations: Vec<SpriteSheetAnimation>,
```

```
    current_animation: u32,
}
//选择索引进行动画的演示
...
if x_speed != 0.0 {
                self.current_animation = 0;
            } else {
                self.current_animation = 1;
            }
//在这里索引0指向移动时的动画，索引1指向跑动时的动画
...
if let Some(current_animation) =
                self.animations.get_mut(self.current_anima
            {
                current_animation.update(context.dt);

                if let Some(sprite) = context
                    .scene
                    .graph
                    .try_get_mut(self.sprite)
                    .and_then(|n| n.cast_mut::<Rectangle>(
                {
                    sprite
                        .material()
                        .data_ref()
                        .set_texture(&"diffuseTexture".int
                        .unwrap();
                    sprite.set_uv_rect(
                        current_animation
                            .current_frame_uv_rect()
                            .unwrap_or_default(),
                    );
                }
            }
 //然后根据选择的索引，进行动画的播放。
```

- 实现角色的攻击

  角色的攻击实现，也是通过按键反应得到攻击属性的改变，然后通过角色攻击属性的布尔值来选择动画的索引值，进行攻击动画的播放，在本人的设定当中，一共有三种攻击方式，通过按键U、I、O进行展示。攻击图片：

在攻击怪物的时候，通过怪物与角色的距离判断和按键的信息结合，若距离接近且有攻击按键输入，则说明发生了攻击，那么会在怪物的函数中进行受攻击的动画演示。

具体代码实现：

```
//攻击按键：
KeyCode::KeyU => {
                            self.attack1 = is_pressed;
                            println!("attack1: {}", is_presse
                        }
                        KeyCode::KeyI => {
                            self.attack2 = is_pressed;
                            println!("attack2: {}", is_presse
                        }
                        KeyCode::KeyO => {
                            self.attack3 = is_pressed;
                            println!("attack3: {}", is_presse
                        }
...
//攻击动画索引
if self.attack1 {
                self.current_animation = 2;
            }
```

```
            if self.attack2 {
                self.current_animation = 3;
            }
            if self.attack3 {
                self.current_animation = 4;
            }
//怪物受攻击判定和演示
if self.target.is_some() {
                let target_position = ctx.scene.graph[self.ta
                let self_position = ctx.scene.graph[ctx.handl
                self.direction = (self_position.x - target_po
                if target_position.metric_distance(&self_posi
                    if self.attacked == true {
                        self.health -= 1;
                        self.current_animation.set_value_and_
                    }
                }
            }
.....
```

PART 3 实现怪物并实现怪物基础AI和动画的实现

- 怪物的创建，怪物的基本创建和玩家人物的基本创建类似，同样也添加了静止时的动画和移动时的动画。具体代码：

```
//怪物的结构体
pub struct Bot {
    rectangle: InheritableVariable<Handle<Node>>,
    ground_probe: InheritableVariable<Handle<Node>>,
    ground_probe_distance: InheritableVariable<f32>,
    ground_probe_timeout: f32,
    speed: InheritableVariable<f32>,
    direction: f32,
    front_obstacle_sensor: InheritableVariable<Handle<Node>>,
```

```
    back_obstacle_sensor: InheritableVariable<Handle<Node>>,
    health: i32,
    attacked: bool,
    dead: bool,
    #[visit(skip)]
    #[reflect(hidden)]
    target: Handle<Node>,
    animations: Vec<SpriteSheetAnimation>,
    current_animation: InheritableVariable<u32>,
}
```

- 怪物的AI设定

  怪物的移动AI

  - 首先，怪物需要进行移动，通过do_move函数实现。

```
fn do_move(&mut self, ctx: &mut ScriptContext) {
        let Some(rigid_body) = ctx.scene.graph.try_get_mut_of
            return;
        };
        let y_vel = rigid_body.lin_vel().y;
        rigid_body.set_lin_vel(Vector2::new(-*self.speed * se
        let Some(rectangle) = ctx.scene.graph.try_get_mut(*se
            return;
        };

        rectangle.local_transform_mut().set_scale(Vector3::ne
            2.0 * self.direction.signum(),
            2.0,
            1.0,
        ));
    }
```

  - 其次由于怪物的移动会受到障碍物和关卡方块阻碍，需要怪物能够识别前方障碍并返回。具体代码实现在has_obstacles函数中实现。

```
fn has_obstacles(&mut self, ctx: &mut ScriptContext) -> bool
        let graph = &ctx.scene.graph;
        let sensor_handle = if self.direction < 0.0 {
            *self.back_obstacle_sensor
        } else {
            *self.front_obstacle_sensor
        };
        let Some(obstacle_sensor) = graph.try_get_of_type::<C
            return false;
        };

        for intersection in obstacle_sensor
            .intersects(&ctx.scene.graph.physics2d)
            .filter(|i| i.has_any_active_contact)
        {
            for collider_handle in [intersection.collider1, i
                let Some(other_collider) = graph.try_get_of_t
                else {
                    continue;
                };

                let Some(rigid_body) = graph.try_get_of_type:
                else {
                    continue;
                };

                if rigid_body.body_type() == RigidBodyType::S
                    return true;
                }
            }
        }

        false
    }
```

- 同时，由于是天空岛，怪物的移动也会受到有无陆地影响，因此需要判断前方是否有陆地。具体代码实现在has_ground_in_front函数中。

```
fn has_ground_in_front(&self, ctx: &ScriptContext) -> bool {
        let Some(ground_probe) = ctx.scene.graph.try_get(*sel
            return false;
        };
        let ground_probe_position = ground_probe.global_posit
        let mut intersections = Vec::new();
        ctx.scene.graph.physics2d.cast_ray(
            RayCastOptions {
                ray_origin: ground_probe_position.into(),
                ray_direction: Vector2::new(0.0, -*self.groun
                max_len: *self.ground_probe_distance,
                groups: Default::default(),
                sort_results: true,
            },
            &mut intersections,
        );

        for intersection in intersections {
            let Some(collider) = ctx.scene.graph.try_get(inte
                continue;
            };

            let Some(rigid_body) = ctx
                .scene
                .graph
                .try_get_of_type::<RigidBody>(collider.parent
            else {
                continue;
            };

            if rigid_body.body_type() == RigidBodyType::Stati
                && intersection
```

```
                    .position
                    .coords
                    .metric_distance(&ground_probe_position)
                    <= *self.ground_probe_distance
            {
                return true;
            }
        }

        false
    }
```

怪物的索敌AI

- 当怪物附近有敌人，也就是玩家存在的时候，怪物需要向玩家靠近，并对玩家发起攻击。索敌的基本原理也就是在玩家和怪物的坐标之间进行距离计算，当到达一定阈值的时候，怪物的移动方向改为玩家方向。而当玩家和怪物之间的距离在一定阈值时，就停下，对玩家开展攻击。具体代码实现：

```
//索敌函数
fn search_target(&mut self, ctx: &mut ScriptContext) {
        let game = ctx.plugins.get::<Game>();
        let self_position = ctx.scene.graph[ctx.handle].globa

        let Some(player) = ctx.scene.graph.try_get(game.playe
            return;
        };
        let player_position = player.global_position();

        let signed_distance = player_position.x - self_positi
        if signed_distance.abs() < 3.0 && signed_distance.sig
            self.target = game.player;
        }
    }
```

- 怪物的移动动画，攻击动画，死亡动画，受攻击动画的实现

```
fn on_update(&mut self, ctx: &mut ScriptContext) {
        if self.dead == false {
            self.search_target(ctx);
            if self.has_obstacles(ctx) {
                self.direction = -self.direction;
            }
            self.ground_probe_timeout -= ctx.dt;
            if self.ground_probe_timeout <= 0.0 {
                if !self.has_ground_in_front(ctx) {
                    self.direction = -self.direction;
                }
                self.ground_probe_timeout = 0.3;
            }
            if self.target.is_some() {
                let target_position = ctx.scene.graph[self.ta
                let self_position = ctx.scene.graph[ctx.handl
                self.direction = (self_position.x - target_po
                if target_position.metric_distance(&self_posi
                    self.speed.set_value_and_mark_modified(1.
                } else {
                    self.speed.set_value_and_mark_modified(0.
                }
            }
            self.do_move(ctx);
            if self.direction != 0.0 {
                self.current_animation.set_value_and_mark_mod
            }
            if self.target.is_some() {
                let target_position = ctx.scene.graph[self.ta
                let self_position = ctx.scene.graph[ctx.handl
                if target_position.metric_distance(&self_posi
                    self.current_animation.set_value_and_mark
                }
            }
            if self.target.is_some() {
```

```
                let target_position = ctx.scene.graph[self.ta
                let self_position = ctx.scene.graph[ctx.handl
                self.direction = (self_position.x - target_po
                if target_position.metric_distance(&self_posi
                    if self.attacked == true {
                        self.health -= 1;
                        self.current_animation.set_value_and_
                    }
                }
            }
        }
        if self.health <= 0 {
            println!("{}", self.health);
            self.current_animation.set_value_and_mark_modifie
            self.stop_move(ctx);
            self.dead = true;
        }
        if let Some(current_animation) = self.animations.get_
            current_animation.update(ctx.dt);

            if let Some(sprite) = ctx
                .scene
                .graph
                .try_get_mut_of_type::<Rectangle>(*self.recta
            {
                sprite
                    .material()
                    .data_ref()
                    .set_texture(&"diffuseTexture".into(), cu
                    .unwrap();
                sprite.set_uv_rect(
                    current_animation
                        .current_frame_uv_rect()
                        .unwrap_or_default(),
                );
            }
```

```
        }
    }
```

当怪物受到角色攻击时，生命值会减少，若生命值减少为0或负，则死亡。