06

Herança

Transcrição

Considerando os problemas listados anteriormente, podemos usar de dois conceitos da orientação a objetos para nos ajudar. *Herança*, para que possamos melhorar um pouco nosso código e conseguir compartilhar características semelhantes entre o jogador e o inimigo, e o *polimorfismo* com a *sobrescrita* de métodos para diferenciar os comportamentos divergentes.

Criaremos então um novo *script* chamado `Personagem.js` que terá o seguinte código:

```
cc.Class({
    extends: cc.Component,

    properties: {
      tiroPrefab: cc.Prefab,
      _direcao: cc.Vec2,

    },

    atirar: function(){
        let disparo = cc.instantiate(this.tiroPrefab);
        disparo.parent = this.node.parent;
        disparo.position = this.node.position;
        disparo.group = this.node.group;

        let componenteTiro = disparo.getComponent("Tiro");
        componenteTiro.direcao = this._direcao;
    },

    tomarDano: function(){

    }

    // use this for initialization
    onLoad: function () {
    },

    // called every frame, uncomment this function to activate update callback
    // update: function (dt) {
    // },
});
```

As propriedades `tiroPrefab` e `direcao` são necessárias para que o personagem se mova e também consiga disparar e indicar a direção do tiro. O método `atirar` também é comum tanto ao jogador quanto ao inimigo. O método `tomarDano` foi criado para podermos diferenciar esse comportamento do jogador e do inimigo.

Ainda precisamos de um ajuste neste código do *script* `Personagem.js` . Ele por hora não está sendo exportado para que possamos utilizá-lo em outros lugares. Faremos isso guardando o resultado da execução do método `Class` da *Cocos* em uma variável e depois exportaremos essa variável com o uso do `module.exports` .

```
let Personagem = cc.Class({
    // todo o código da classe
});

module.exports = Personagem;
```

Agora precisamos fazer com que os *scripts* `Jogador.js` e `Inimigo.js` reusem este código que isolamos no *script* `Personagem.js` e também ter o cuidado de remover o código que se repete. Neste caso as propriedades `tiroPrefab` e `direcao`, assim como o método `atirar`. Vejamos como fica a classe do *script* `Jogador.js`.

```
let Personagem = require("Personagem");

cc.Class({
    extends: Personagem,

    properties: {
        _acelerando: false,
        velocidade: 200,
    },

       // use this for initialization
    onLoad: function () {
        cc.SystemEvent.on(cc.SystemEvent.EventType.KEY_DOWN, this.teclaPressionada, this);
        cc.SystemEvent.on(cc.SystemEvent.EventType.KEY_UP, this.teclaSolta, this);

        let canvas = cc.find("Canvas");
        canvas.on("mousemove", this.mudarDirecao, this);
        canvas.on("mousedown", this.atirar, this);

        cc.director.getCollisionManager().enabled = true;
    },

    mudarDirecao: function(event){
        let posicaoMouse = event.getLocation();
        posicaoMouse = new cc.Vec2(posicaoMouse.x, posicaoMouse.y);

        let direcao = posicaoMouse.sub(this.node.position);
        direcao = direcao.normalize();

        this._direcao = direcao;
    },

    teclaPressionada: function(event){
        if(event.keyCode == cc.KEY.a){
            this._acelerando = true;
        }
    },

    teclaSolta: function(event){
        if(event.keyCode == cc.KEY.a){
            this._acelerando = false;
        }
    },

    // called every frame, uncomment this function to activate update callback
```

```
        update: function (dt) {
         if(this._acelerando){
             let deslocamento = this._direcao.mul(this.velocidade * dt);
             this.node.position = this.node.position.add(deslocamento);
         }
    },
});
```

A principal mudança além da remoção do código, foi a importação do código presente no `Personagem.js` usando a função `require` informando o nome do arquivo.

```
let Personagem = require("Personagem");
```

Além disso, alteramos a linha de extensão de classe de `cc.Component` para `Personagem`. Isso faz com que a linhagem de herança seja um pouco mais extensa. Afinal, o jogador agora herda de personagem, que por sua vez herda de `Component`. O *script* do inimigo também precisa herdar este código. Assim teremos:

```
let Personagem = require("Personagem");

cc.Class({
    extends: Personagem,

    properties: {
      _alvo: cc.Node,
      velocidade: 50,
      tempoAtaque: 1,
    },

    // use this for initialization
    onLoad: function () {
        this._alvo = cc.find("hero");
        this.schedule(this.atirar, this.tempoAtaque);
    },

    atirar: function(){
        let disparo = cc.instantiate(this.tiroPrefab);
        disparo.parent = this.node.parent;
        disparo.position = this.node.position;

        disparo.group = this.node.group;

        let componenteTiro = disparo.getComponent("Tiro");
        componenteTiro.direcao = this._direcao;
    },

    mudarDirecao: function(){
        let direcao = this._alvo.position.sub(this.node.position);
        direcao = direcao.normalize();
        this._direcao = direcao;
    },

    // called every frame, uncomment this function to activate update callback
    update: function (dt) {
        this.mudarDirecao();
```

```
        let deslocamento = this._direcao.mul(this.velocidade * dt);
        this.node.position = this.node.position.add(deslocamento);
    },
});
```

A ideia agora é fazer com que cada personagem reaja de forma diferente ao ser atingido pelo tiro. O método `tomarDano` no *script* `Inimigo.js` fará simplesmente o objeto ser destruído na cena. Algo que a gente já sabe fazer.

```
tomarDano: function(){
    this.node.destroy();
}
```

Já no `Jogador.js` , fará ele perder pontos de vida. Estes pontos precisam ser recebidos por parâmetro.

```
tomarDano: function(dano){
    this.vida -= dano;
}
```

E no `Tiro.js` paramos de simplesmente destruir o objeto da cena, para executar o método `tomarDano` do componente `Personagem` do objeto que colidiu com ele. No método `onCollisionEnter` teremos:

```
onCollisionEnter:function(outro, eu){
    let personagem = outro.getComponent("Personagem");
    personagem.tomarDano(2);
    eu.node.destroy();
}
```

O parâmetro `2` na chamada do método `tomarDano` será ignorado pelo inimigo e o teremos destruído. Já no caso do jogador, ao ser atingido, reduzi-se seus pontos de vida. Lembre-se de criar a propriedade `vida` com o valor `100` no `Jogador.js` para que tudo funcione como esperado.

Apesar de parecer tudo certo, ainda temos um problema. Caso um tiro atinja outro tiro, nosso jogo irá travar porque o componente `Personagem` não existe para o objeto tiro. Assim teremos erros ao tentar executar o `tomarDano` . Um tiro não toma danos. Caso um tiro colida com o outro, os dois se destroem. Podemos codificar esse comportamento verificando se a variável `personagem` é nula. Caso seja, um tiro colidiu com o outro. Assim teremos:

```
onCollisionEnter:function(outro, eu){
    let personagem = outro.getComponent("Personagem");
    if(personagem != null)        {
        personagem.tomarDano(2);
    }else{
        outro.node.destroy();
    }
    eu.node.destroy();
},
```

Agora nosso jogo funciona perfeitamente. Alguns detalhes ainda estão faltando, mas logo serão resolvidos. Por exemplo, estamos causando dano ao jogador, mas não temos nenhum *feedback* disso em nosso jogo. Algo que trabalharemos adiante.