目录

[一、 图片预加载脚本 1](#_Toc22540)

[二、 背景图无缝轮播脚本 1](#_Toc13214)

[三、 栅栏图无缝轮播脚本 3](#_Toc21965)

[四、 小鱼动画脚本 4](#_Toc10046)

[五、 障碍物冰柱动画脚本 7](#_Toc18144)

[六、 游戏结束事件脚本 10](#_Toc14410)

[七、 游戏控制器脚本 12](#_Toc3725)

2. 图片预加载脚本

在‘小鱼海棠’游戏中，各个游戏的主要元素都是以图片为基础，通过将图片不断地绘制拼接，加上定时器函数，才构成一个生动的动画。但是游戏中图片的引用并不是普通页面中的img标签一样，直接给标签定义好src路径，让其慢慢加载。在游戏中，并不是一个一个元素的加载使用，而是所有图片预加载完毕一起调用，所以要封装一个图片预加载脚本。代码如下。

function loadImg(src, fn) {

var key;

var tempImg, resultImg = {};

var loadedTotal = 0,

total = 0;

for(key in src) {

// 求要加载图片的总数

total++;

// 创建对应的img

tempImg = new Image();

tempImg.src = src[key];

// 存储起来

resultImg[key] = tempImg;

// 为了保证所有img加载完毕后，执行回调，必须要监听每一个img的onload事件

tempImg.onload = function() {

if(++loadedTotal >= total) {

fn(resultImg);

}

};

}

}

1. 背景图无缝轮播脚本

在背景图片已经加载完毕的情况之下，需要封装一个构造函数来生成游戏背景的对象，实现游戏背景图片无缝轮播，使页面动画更形象生动。代码如下。

(function(w) {

/\*

\* constrctor { Sky } 游戏背景

\* param { options } 背景参数配置

\* patam { options.ctx: Context } 绘图环境

\* patam { options.img: Image } 背景图片

\* \*/

function Sky(options) {

// 每创建一个背景，那么总数自增一次

Sky.len++;

this.ctx = options.ctx;

this.img = options.img;

// 一张背景的宽和高

this.width = options.img.width;

this.height = options.img.height;

// 背景的x轴坐标按照数量动态计算

// 例：第一个背景x轴坐标为0，第二个背景x轴坐标为一个背景宽度

this.x = this.width \* (Sky.len - 1);

this.y = 0;

// 背景每秒运行的速度

this.speedSecond = 100;

}

// 记录创建背景对象的数量

Sky.len = 0;

// 替换原型

Sky.prototype = {

constructor: Sky,

// 绘制背景

draw: function() {

this.ctx.drawImage(this.img, this.x, this.y);

},

// 更新背景下一次绘制时的数据

update: function(delaySecond) {

// 背景做匀速运动

// 行走的总路程 = 初速度 \* 时间

// 因为背景向左运动，所以背景最新的x轴坐标 = 当前x轴 - 运动的总路程

this.x -= this.speedSecond \* delaySecond;

// 如果背景走出画布，那么向右拼接，实现无缝滚动效果

if(this.x < -this.width) {

this.x += this.width \* Sky.len;

}

}

};

w.Sky = Sky;

}(window));

1. 栅栏图无缝轮播脚本

在页面下部绘制栅栏元素，封装一个栅栏元素的构造函数，在函数的原型对象中定义绘制栅栏的方法和栅栏无缝滚动的方法。多次调用函数，实现栅栏元素拼接占满整个屏幕，同事栅栏是不断运动的，设置栅栏最新的位置 = 当前位置 - 总运动距离( 速度 \* 时间 )，当栅栏走出屏幕时，使其横坐标加上栅栏图片的宽度乘以申明函数的次数，为新的栅栏横坐标，实现向右拼接，实现栅栏的无缝滚动。代码如下。

(function( w ) {

/\*

\* constrctor { Land } 游戏大地

\* param { options } 背景参数配置

\* patam { options.ctx: Context } 绘图环境

\* patam { options.img: Image } 大地图片

\* \*/

function Land( options ) {

// 没创建一个大地就自增一下总数

Land.len++;

this.ctx = options.ctx;

this.img = options.img;

// 大地的宽和高

this.width = this.img.width;

this.height = this.img.height;

// 大地的坐标

this.x = this.width \* (Land.len - 1);

this.y = this.ctx.canvas.height - this.height;

// 大地每秒运行的速度

this.speedSecond = options.speedSecond;

}

// 记录创建了多少大地

Land.len = 0;

// 置换原型

Land.prototype = {

constructor: Land,

// 绘制大地

draw: function() {

this.ctx.drawImage( this.img, this.x, this.y );

},

// 更新大地下一次绘制时的数据

update: function( delaySecond ) {

// 大地最新的位置 = 当前位置 - 总运动距离( 速度 \* 时间 )

this.x -= this.speedSecond \* delaySecond;

// 当大地走出画布时，向右拼接，实现无缝滚动

if( this.x < -this.width ) {

this.x += this.width \* Land.len;

}

}

};

w.Land = Land;

}( window ));

1. 小鱼动画脚本

在小鱼构造函数的原型对象中定义小鱼游动时旋转动画，先取小鱼的中心点坐标，在页面中根据小鱼图片一排的帧数绘制小鱼游动的动画，接下来定义小鱼每时刻旋转的角度为每单位的旋转角度乘以速度，因为预先定义了小鱼旋转的最大角度最小角度，所以绘制旋转的小鱼时就不会担心小鱼会无限制的旋转下去。旋转动画定义好之后，来定义小鱼下落的动画。根据物理公式，小鱼下降的总路程 = 初始速度 \* 时间 + 加速度 \* 时间 \* 时间 / 2，需要每时刻更新小鱼的位置，小鱼的最新的位置 = 当前位置 + 总路程，根据CSS3提供的translate方法，就可以让小鱼不断地下降。最后，给页面绑定一个点击事件，每当页面点击的时候，给速度赋值为-150，因为定义的小鱼游动动画、下落动画、旋转动画都与速度息息相关，每次点击页面小鱼都会恢复初始状态。具体代码如下。

(function( w ) {

/\*

\* constrctor {Fish } 鱼

\* param { options } 参数配置

\* patam { options.ctx: Context } 绘图环境

\* patam { options.img: Image } 鱼图片

\* patam { options.widthFrame: nunber } 一排有多少帧

\* patam { options.heightFrame: nunber } 一排有多少帧

\* \*/

function Fish( options ) {

this.ctx = options.ctx;

this.img = options.img;

// 小鱼的宽和高

this.width = this.img.width / options.widthFrame;

this.height = this.img.height / options.heightFrame;

// 小鱼的坐标

this.x = this.width;

this.y = this.height;

// 小鱼每秒运行的初始速度

this.speedSecond = 100;

this.a = 300;

// 当前小鱼飞翔的第几个动作

this.index = 0;

// 定义小鱼1单位速度对应的旋转角度

this.unitAngle = 0.5;

// 小鱼旋转的角度

this.rotateAngle = 0;

// 小鱼旋转的最大角度

this.maxRotateAngle = 40;

// 小鱼旋转的最小角度

this.minRotateAngle = -30;

this.bind();

}

// 置换原型

Fish.prototype = {

constructor: Fish,

// 绘制鱼

draw: function() {

this.ctx.save();

// 平移到小鱼的中心点

var fishCoreX = this.x + this.width / 2;

var fishCoreY = this.y + this.height / 2;

this.ctx.translate( fishCoreX, fishCoreY );

// 旋转的角度 = 1单位速度旋转的角度 \* 速度

this.rotateAngle = this.unitAngle \* this.speedSecond;

this.rotateAngle = this.rotateAngle > this.maxRotateAngle? this.maxRotateAngle: this.rotateAngle;

this.rotateAngle = this.rotateAngle < this.minRotateAngle? this.minRotateAngle: this.rotateAngle;

// 旋转指定的弧度

this.ctx.rotate( util.angleToRadian( this.rotateAngle ) );

// 绘制旋转的小鱼

this.ctx.drawImage( this.img,

this.width \* this.index, 0, this.width, this.height,

-this.width / 2, -this.height / 2, this.width, this.height);

this.ctx.restore();

},

// 更新下一次鱼的数据

update: function( delaySecond ) {

this.index = ++this.index % 3;

// 总路程 = 初始速度 \* 时间 + 加速度 \* 时间 \* 时间 / 2

// 小鸟最新的位置 = 当前位置 + 总路程

this.y += this.speedSecond \* delaySecond + this.a \* delaySecond \* delaySecond / 2;

// 更新下落的速度，因为加速度的存在，下落越来越快

// 新的速度 = 初始速度 + 加速度 \* 时间

this.speedSecond += this.a \* delaySecond;

},

// 绑定点击事件，点击小鱼上飞

bind: function() {

var self = this;

this.ctx.canvas.addEventListener( 'click', function() {

self.speedSecond = -150;

});

}

}

w.Fish = Fish;

}( window ));

1. 障碍物冰柱动画脚本

在封装障碍物冰柱的构造函数时，首先要设置调用函数时传入的形参：预加载完毕的障碍物冰柱图片、绘图环境Canvas对象、上下障碍物冰柱的间距、左右障碍物冰柱的间距、障碍物冰柱的最小高度、障碍物冰柱每秒的运行速度、障碍物冰柱每秒的加速度。在构造函数的原型对象中来定义绘制障碍物冰柱的方法和障碍物无缝滚动的方法。首先计算上部障碍物冰柱y轴的坐标，创建障碍物冰柱对象的时候，需要随机生成障碍物冰柱的高，然后根据Canvas对象的高和上部障碍物的伸出的高度计算下部障碍物的y轴坐标。同时障碍物走出画布的时候，还需要再次随机生成管道的高，在绘制每一个冰柱的同时要绘制冰柱的三角形轮廓。

实现障碍物冰柱无缝轮播的方法和绘制栅栏一样，唯一不同的是为了增加游戏难度，给障碍物冰柱的运动添加了加速度，速度每刻都在刷新，会越来越快。具体代码如下。

(function( w ) {

/\*

\* constructor { Pipe } 管道构造函数

\* param { options: Object } 参数配置

\* param { options.ctx: Context } 绘制上下文

\* param { options.imgDown: Image } 口朝下的管道，在画布的上面

\* param { options.imgUp: Image } 口朝上的管道，在画布的下面

\* param { options.pipeTBSpace: number } 上下管道之间的间隔

\* param { options.pipeLRSpace: number } 管道与管道之间的间隔

\* param { options.minHeight: number } 管道最小高度

\* param { options.maxHeight: number } 管道最大高度

\* \*/

function Pipe( options ) {

// 记录管道创建的总数

Pipe.len++;

this.ctx = options.ctx;

this.imgDown = options.imgDown;

this.imgUp = options.imgUp;

// 管道图像的宽和高

this.imgWidth = this.imgDown.width;

this.imgHeight = this.imgDown.height;

// 管道上下间距和管道左右间距

this.pipeTBSpace = options.pipeTBSpace || 150;

this.pipeLRSpace = options.pipeLRSpace;

// 管道的坐标

this.x = 400 + (this.imgWidth + this.pipeLRSpace) \* (Pipe.len - 1);

this.y = 0;

// 管道最小高度和最大高度

this.minHeight = options.minHeight;

this.maxHeight = options.maxHeight;

// 管道每秒运行的速度

this.speedSecond = options.speedSecond;

// 管道每秒的加速度

this.a = options.a;

this.computePipeY();

}

// 记录管道的数量

Pipe.len = 0;

// 置换原型

Pipe.prototype = {

constructor: Pipe,

// 计算上下管道y轴的坐标

// 创建管道对象的时候，需要随机生成管道的高，然后计算上下管道的y轴坐标

// 以后管道走出画布的时候，还需要再次随机生成管道的高，然后计算上下管道的y轴坐标

computePipeY: function() {

// 随机生成上管道的高度

var pipeUpViewHeight = util.random( this.minHeight, this.maxHeight );

// 上管道的Y轴坐标

this.downY = pipeUpViewHeight - this.imgHeight;

// 下管道的Y轴坐标

this.upY = pipeUpViewHeight + this.pipeTBSpace;

},

// 绘制管道

draw: function() {

this.ctx.drawImage( this.imgDown, this.x, this.downY );

this.ctx.drawImage( this.imgUp, this.x, this.upY );

this.drawPath();

},

// 绘制管道对应的冰柱路径

drawPath: function() {

this.ctx.moveTo(this.x,this.downY);

this.ctx.lineTo(this.x+this.imgWidth/2,this.downY+this.imgHeight);

this.ctx.lineTo(this.x+this.imgWidth,this.downY)

this.ctx.moveTo(this.x+this.imgWidth/2,this.upY);

this.ctx.lineTo(this.x, this.upY+this.imgHeight);

this.ctx.lineTo(this.x+this.imgWidth,this.upY+this.imgHeight);

// this.ctx.rect( this.x, this.downY, this.imgWidth, this.imgHeight );

// this.ctx.rect( this.x, this.upY, this.imgWidth, this.imgHeight );

},

// 更新下一次管道绘制时的数据

update: function( delaySecond ) {

// 下一次绘制的x坐标 = 当前坐标 - 总路程

this.x -= this.speedSecond \* delaySecond + this.a \* delaySecond \* delaySecond / 2;

// 根据时间和加速度，更新速度

this.speedSecond += this.a \* delaySecond;

// 如果管道走出画布，向右拼接

if( this.x < -this.imgWidth ) {

this.x += (this.imgWidth + this.pipeLRSpace) \* Pipe.len;

//走出之后再计算，再绘制

this.computePipeY();

}

}

};

w.Pipe = Pipe;

}( window ));

1. 游戏结束事件脚本

游戏结束事件脚本的实现思路是在小鱼超出游戏界面或触碰到障碍物冰柱或触碰到栅栏时游戏结束，绘制一个占满整个屏幕的矩形框，颜色填充为半透明。在矩形框上使用Canvas提供的fillText()方法，绘制文字GAMEOVER!!!，在文字下面绘制两个按钮，但是在Canvas并不像传统的页面元素一样，按钮由HTML元素构成或者由javascript动态生成，当前Canvas并没有提供生成元素的方法，所以要模拟按钮，绘制两个矩形框，通过判断鼠标点击时的坐标是否在矩形框中，来添加对应的响应事件对象，进而跳出当前页面。具体代码如下。

( function( w ) {

/\*

\* constructor { OberScene } 游戏结束场景

\* param { options } 参数配置

\* param { options.ctx: Context } 绘图环境

\* param { }

\* \*/

function OverScene( options ) {

this.ctx = options.ctx;

// 重新开始按钮的宽和高

this.btnWidth = 300;

this.btnHeight = 80;

this.btnX = 0;

this.btnY = 0;

}

util.extend( OverScene.prototype, {

// 绘制游戏结束场景

run: function() {

this.ctx.save();

// 游戏中心坐标

var gameCoreX = this.ctx.canvas.width / 2;

var gameCoreY = this.ctx.canvas.height / 2;

// 绘制一个遮盖层

this.ctx.fillStyle = 'rgba( 0, 0, 0, 0.5 )';

this.ctx.fillRect( 0, 0, this.ctx.canvas.width, this.ctx.canvas.height );

// 绘制文字

this.ctx.font = '500 50px 微软雅黑';

this.ctx.textAlign = 'center';

this.ctx.textBaseline = 'bottom';

this.ctx.fillStyle = 'orange';

this.ctx.fillText( 'GAMEOVER!!!', gameCoreX, gameCoreY - 180 );

// 绘制按钮

this.ctx.strokeStyle = 'deeppink';

this.ctx.fillStyle = 'deeppink';

this.ctx.lineWidth = 4;

this.ctx.textBaseline = 'middle';

this.btnX = gameCoreX - this.btnWidth;

this.btnY = gameCoreY - this.btnHeight / 2 + 40;

this.ctx.strokeRect( this.btnX, this.btnY, this.btnWidth, this.btnHeight );

this.ctx.fillText( '重新开始', this.btnX + this.btnWidth / 2, this.btnY + this.btnHeight / 2 );

this.ctx.beginPath();

this.ctx.strokeRect( this.btnX+400, this.btnY, this.btnWidth, this.btnHeight );

this.ctx.fillText( '选择难度', this.btnX + this.btnWidth / 2+400, this.btnY + this.btnHeight / 2 );

this.ctx.restore();

},

// 重新开始按钮点击事件

bind: function() {

var self = this;

this.ctx.canvas.addEventListener( 'click', function( e ) {

// 求鼠标点击时相对于画布的路径

var x= e.pageX - this.offsetLeft;

var y = e.pageY - this.offsetTop;

// 画按钮的矩形路径

self.ctx.beginPath();

self.ctx.rect( self.btnX, self.btnY, self.btnWidth, self.btnHeight );

self.ctx.beginPath();

self.ctx.rect( self.btnX+400, self.btnY, self.btnWidth, self.btnHeight );

// 如果在，刷新页面重新开始游戏

if(x>self.btnX&&x<self.btnX+self.btnWidth&&y>self.btnY&&y<self.btnY+self.btnHeight ) {

location.reload();

}

//如果在，重新选游戏难度

if( x>self.btnX+400&&x<self.btnX+400+self.btnWidth&&y>self.btnY&&y<self.btnY+self.btnHeight ) {

window.location.href='../../index.html';

}

});

}

} );

w.OverScene = OverScene;

}( window ));

1. 游戏控制器脚本

游戏控制器脚本的中心功能调用封装的游戏元素函数，创建游戏运行时所需的所有角色，支配所有的角色执行其动画效果，判断游戏进程中是否触发小鱼死亡事件，若触发，执行死亡事件回调函数。页面中所有的游戏元素动画都是基于定时器的时间差不断地快速刷新页面，快到肉眼难以分辨，视觉上形成一个连续动画的效果。在时间间隙中不断地判断小鱼是否触发死亡事件，如若死亡，立即关闭定时器，所有的游戏动画静止，执行游戏结束回调函数。具体代码如下。

(function(w) {

/\*

\* constructor { GameScene } 游戏运行的场景对象

\* param { options } 对象配置

\* param { options.ctx: Context } 绘图上下文

\* param { options.imgs: Object } 创建角色所需的所有图像

\* \*/

function GameScene(options) {

this.ctx = options.ctx;

this.imgs = options.imgs;

// 存储游戏运行时所需的所有角色

this.roles = [];

// 小鱼死亡时，要执行的所有回调函数

this.FishDieCallbackList = [];

this.createRoles();

}

/\*

\* 给游戏场景对象的原型添加方法：

\* 1、创建角色的方法

\* 2、支配所有角色的方法

\* 3、小鱼是否死亡的方法（死亡要通知游戏主函数，解决方式一会再说）

\* \*/

util.extend(GameScene.prototype, {

// 创建该场景所需的角色

createRoles: function() {

var speedSecond =parseInt(sessionStorage.getItem("speed"));

var pipeTBSpace=parseInt(sessionStorage.getItem("tb")) ;

var pipeLRSpace=parseInt(sessionStorage.getItem("lr"));

var a=parseInt(sessionStorage.getItem("a"));

var minHeight=parseInt(sessionStorage.getItem("min"));

// 创建2个背景对象

for(var i = 0; i < 2; i++) {

this.roles.push(new Sky({

ctx: this.ctx,

img: this.imgs.sky,

speedSecond: speedSecond

}));

}

// 创建7个管道对象

for(var i = 0; i < 7; i++) {

this.roles.push(new Pipe({

ctx: this.ctx,

imgDown: this.imgs.pipeDown,

imgUp: this.imgs.pipeUp,

a: a,

speedSecond: speedSecond,

pipeTBSpace: pipeTBSpace,

pipeLRSpace: pipeLRSpace,

minHeight: minHeight,

maxHeight: this.ctx.canvas.height - this.imgs.land.height - minHeight - pipeTBSpace

}));

}

// 创建4个大地对象

for(var i = 0; i < 4; i++) {

this.roles.push(new Land({

ctx: this.ctx,

speedSecond: speedSecond,

img: this.imgs.land

}));

}

// 创建1个计时器对象

this.roles.push(new Timer({ ctx: this.ctx }));

// 创建1个鱼对象

this.roles.push(new Fish({ ctx: this.ctx, img: this.imgs.fish, widthFrame: 3, heightFrame: 1 }));

},

// 让所有的角色依据游戏的延迟时间进行响应

run: function(delaySecond) {

this.roles.forEach(function(role) {

role.draw();

role.update(delaySecond);

});

// 每次绘制完毕后，看看小鱼有没有死亡，如果死亡了，

// 执行所有的小鱼死亡回调

if(this.isFishDie()) {

this.runFishDieCbk();

}

},

// 判断小鱼是否死亡

isFishDie: function() {

var fish = this.roles[this.roles.length - 1];

var fishCoreX = fish.x + fish.width / 2;

var fishCoreY = fish.y + fish.height / 2;

if(fishCoreY < 0 || fishCoreY > (this.ctx.canvas.height - this.imgs.land.height) ||

this.ctx.isPointInPath(fishCoreX, fishCoreY)) {

return true;

}

return false;

},

// 执行小鱼死亡时所有的回调

runFishDieCbk: function() {

this.FishDieCallbackList.forEach(function(fn) {

fn();

});

},

// 添加小鱼死亡的回调

addFishDieCbk: function(fn) {

this.FishDieCallbackList.push(fn);

}

});

w.GameScene = GameScene;

}(window));