多媒体技术基础

BY 沈小双

软工1406—201426810718

1 设计文档

1.1 整体说明

1.1.1 系统类型

系统类型	小游戏
开发所用操作系统	Linux 4.4.0-24-generic Ubuntu 16.04LTS GNU/Linux
开发语言	Racket
IDE	DrRacket 6.5

表格 1.

注意 1. 源代码必须使用DrRacket6.5或更新的版本打开，请勿使用普通的文本编辑器打开。在文档中会有大部分的关键代码。

1.1.2 设计目的

这是一个模仿chrome浏览器离线时出现的小恐龙游戏的小游戏。构思这个游戏一开始是因为和室友在近代史课上无聊的消遣活动，也因为自己从没有写过什么游戏相关的东西。这是一个十分简单的游戏，也是我在这方面的处女作。所以趁着多媒体课的机会将其从纸上实现为代码。其实一开始的构思十分不完全，写了之后才发现之前构思的不足。游戏并没有用到任何的游戏引擎，完全由racket实现。总之，Just for fun。

当然这次的游戏制作让我了解了,游戏中的碰撞检测大致使如何实现的.同时也多谢老师的指正让我意识到这款游戏本身游戏性的不足.

1.1.3 文档结构

TheGame.rkt 游戏主程序,运行次脚本就可以

test.rkt 测试脚本

help_functions.rkt 辅助函数

document.pdf 说明文档

1.1.4 系统名称

小恐龙

1.1.5 需求分析

这个游戏的用户估计也就只有我自己，所以就个人而言这是一个模仿，基本达到和原本游戏80%的相似度，如果游戏性更高当然最好。

- 接受用户键盘输入，实现操作

- 随机生成障碍物
- 实现计分功能,这是游戏玩的好坏的指标
- 实现每次起跳的声音
- 进行碰撞检测，这几乎是这个游戏的设计中对我而言最为困难的部分

1.1.6 系统模块

- 按键处理模块

  响应用户的按键,并且根据按键处理

- 时间处理模块

  由时间线,计算出下一幕的游戏状态

- 障碍物随机生成模块

  随机生成障碍物加入到障碍物列表中

- 碰撞判断模块

  判断恐龙是否和障碍物碰撞,使用了改进的包围盒的碰撞检测,效率高,判断精确

- 启动模块

  设定启动参数,游戏的初始状态

- 绘制模块

  根据输入的游戏状态绘制出图片

- 计分模块

  顾名思义,根据走过的距离计分

- 综合罗辑框架

  处理时间,键盘,绘图模块之间的罗辑

```
(big-bang game-state
            [on-draw drawer];;绘制
            [on-tick time-render 0.02 ];;时间
            [on-key key-render];;相应键盘事件
            [stop-when last-world?]
            )
```

1.2 制作说明

1.2.1 制作流程

1. 搜集,制作游戏素材(对于碰撞检测有特殊要求)

2. 编写游戏罗辑代码

3. 调整游戏参数设置,使难度适中

4. 测试

1.2.2 关键代码与介绍

游戏总体罗辑,使用game-state来描述游戏的状态,该数据字段包含了所有绘制,罗辑处理时的游戏信息.键盘时间处理函数会改变这个数据结构中的内容,以达到对游戏状态的修改,绘制函数也是根据当前的游戏状态信息绘制出游戏的画面.

```
(big-bang game-state
            [on-draw drawer];;绘制画面
            [on-tick time-render 0.02 ];;时间处理,得到下一幕的游戏状态
            [on-key key-render];;处理键盘事件
            [stop-when last-world?];;判断游戏是否已经结束
            )
```

碰撞算法,这是游戏的逻辑设计上的难点.从一开始的简单的矩形包围盒的碰撞检测到改进的精细的碰撞检测.

i. 过滤掉明显不可能相撞的障碍物,由于障碍距离恐龙的距离太远,不肯能相撞,通过判断障碍物的x坐标过滤掉大部分的障碍物

```
(define (out-picture? obj)
  (let ([x (posn-x (third obj))])
    (cond
      [(< x 0) #f]
      [else #t])))

;;过滤障碍物列表中超过画布左侧的物体
(define (filter-objects obj)
  (filter out-picture? obj))
```

ii. 进行简单的包围盒碰撞检测,将恐龙和障碍物都看做是两个矩形,通过四条边的关系可以判断有无重叠部分,如果没有重叠则未检测到碰撞,如果检测到重叠部分则再对进行判断重叠部分.(由于图片中有存在空白部分,空白部分重叠不应被判断为碰撞)

例 2.


以上两幅图片,第一幅是简单的包围盒检测判断到的碰撞,第二个是改进后看到的碰撞检测检测到的碰撞,在改进后第一幅图片也被判定为未碰撞.

```
(define (pounch? d obj)
  (local(;;两个坐标和两个图片,确定是否碰撞,p1为恐龙原图
         (define (pounch? x1 y1 x2 y2 p1 p2)
           (local ((define w1 (image-width p1))
                   (define h1 (image-height p1))
                   (define w2 (image-width p2))
```

```
                    (define h2 (image-height p2))
                    (define lp1 (- x1 (/  w1 2)))
                    (define rp1 (+ x1 (/  w1 2)))
                    (define tp1 (- y1 (/  h1 2)))
                    (define bp1 (+ y1 (/  h1 2)))
                    (define lp2 (- x2 (/  w2 2)))
                    (define rp2 (+ x2 (/  w2 2)))
                    (define tp2 (- y2 (/  h2 2)))
                    (define bp2 (+ y2 (/  h2 2)))
                    (define xl (list lp1 rp1 lp2 rp2))
                    (define yl (list tp1 bp1 tp2 bp2)))
                (cond
                  ;;由四条边进行判断部分
                  [(rp1 . < . lp2) #f]
                  [(lp1 . > . rp2) #f]
                  [(tp1 . > . bp2) #f]
                  [(bp1 . < . tp2) #f]
                  [else (real-touch? lp1 tp1 xl yl p1 (place-image p2 (- x2
(- x1 (/ w1 2))) (- y2 (- y1 (/ h1 2))) p1))] ;;加入像素点的检测
                  )))
          (define x1 (posn-x (DS-posn d)));;恐龙的坐标
          (define y1 (posn-y (DS-posn d)))
          (define x2 (posn-x (third obj)));;障碍物的坐标
          (define y2 (posn-y (third obj)))
          (define p1 (choose-dinasor d));;恐龙图
          (define p2 (second obj));;障碍物图
          )
    (pounch? x1 y1 x2 y2 p1 p2)))
```

iii. 根据坐标计算得出重叠部分的图片的相应坐标,并对局部进行碰撞检测,这里避免了对整个矩形进行检测极大的减少了碰撞检测的时间消耗.

```
(define (real-touch? lp1 tp1 l1 l2 po p)
  (local (
          (define xl  (take (drop (sort l1 <) 1) 2));;x-list 重合的x的坐标
          (define yl  (take (drop (sort l2 <) 1) 2));;类上
          (define w   (inexact-> xact(floor(- (second xl) (first
xl)))));;宽度
          (define h   (inexact-> xact(floor(- (second yl) (first yl)))))
          (define s-x (inexact-> xact(floor(- (first xl)
lp1))));;start-x
          (define s-y (inexact-> xact(floor(- (first yl) tp1))))
          (define e-x (+ s-x w))
          (define e-y (+ s-y h)))
    (touch? po p s-x s-y e-x
e-y));;计算得到相应的坐标值后交由touch?进行局部检测
```

Acknowledgments. 已知两个矩形(平行或垂直正放置与坐标系)各自的两个对顶角的坐标,共四个点的坐标.如果两个矩形相交那么易知相交矩形的对顶角的坐标.因为相交部分必然是中间部分,例如已知四个点$(X_i, Y_i), i \in \{1, 2, 3, 4\}$,那个相交矩形的坐标X or Y 必然不是$X_i$中的最大或最小值,那么只剩下中间的值,Y同理.

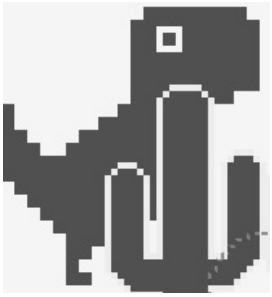
iv. 进行精细的碰撞检测

```
(define (color->n p x y )
  (let([P (get-pixel-color x y p)])
    (cond
      [(or (color=? (color 247 247 247 255)  P)
           (color=? (color 255 255 255 255)  P)) 3];;白
      [(or (color=? (color 255 255 255 0) P)
           (color=? (color 0 0 0 0)       P)) 0];;透明
      [(or (color=? (color 83 83 83 255) P)
           (color=? (color 84 84 84 255) P)) 1];;黑
      [else  (display P)(error "未知颜色")])))

(define (to a b)
  (cond
    [(= a b) (cons b '())]
    [(a . > . b) (error "a 比b大")]
    [else (cons a (to (+ a 1) b ))]))

(define (img->list/n p x0 y0 x y)
  (for*/list ([i ( x0 . to . x)]
              [j ( y0 . to . y)])
    (color->n p i j)))
```

```
(define (touch? p1 p2 s-x s-y e-x e-y)
  (local ((define p1-list (img->list/n p1 s-x s-y e-x e-y))
          (define p2-list (img->list/n p2 s-x s-y e-x e-y))
          (define (foo l1 l2)
            (cond
              [(and (empty? l1) (empty? l2)) #f]
              [(and (= 1 (first l1))
                    (= 3 (first l2))) #t]
              [else (foo (rest l1) (rest l2))])))
    (foo p1-list p2-list)))
```

在游戏的图片被使用之前先采用了,特殊的处理方式例如以下的仙人掌

可以发现,在仙人掌的周围还增加了一圈白色的像素包围,由于游戏的背景本身就是白色所以不会影响游戏的观感,但是在发生碰撞的时候,这一圈白色的像素可以作为碰撞检测的标志.以下是发生碰撞时的图片



可以发现恐龙原来黑色的一部分被障碍物挡住了,由黑色变成白色,所以检测的时候只要用原始的图片和需要检测的图片进行对比,如果存在由黑色变成白色的位置则可以断定发生了碰撞.

这样当遍历完成都没有找到这样的点就认为没有发生碰撞,算法最大的耗时正比与像素区域的像素点的个数一般不超过100,算法复杂度为O(n),如果发生了碰撞则需要的时间更短因为不用再遍历剩下的区域.

关于地面的移动,这是游戏中一个比较tricky的地方,地面一张有限的图片,如何让玩家感觉这是一个连续的永无止境的图片

具体代码

```
;;ground=>ground
;;移动地面
(define(render-ground-by-time ground  speed)
  (let [(x (posn-x (second ground)))]
    (list (first ground) (make-posn
                          (cond
                            [(x . < . (- 0 (* 1/2 canvas-width))) (* 3/2 canvas-width
                            [else (- x speed)])
                          ground-y))))
```

原理说明:

举例现在有一条纸带上面写着ABCD,如何让它循环起来,每次可以读出两个字母.不可以从头开始(应为A左边为空,反应的图片上就是玩家会看到消失的地面),所以我的处理方式是,增加一条一模一样的纸带,所以我们就有了ABCDABCD,你可以从B开始读,读到第二条纸带上的A时跳回到第一条纸带上,这样就可以得到无限循环的BCDABCDABCDA.

1.2.3 各个游戏模块的代码说明

- 时间处理

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;时间函数
(define (time-render state)
  (cond
    ;[(last-world? state) state]
    [(game-waiting? state) state];;等待中?
    [(game-running? state) (next-state-by-time state)];;在跑? 计算下一幕的游戏状态
    [else state]))    ;记得写else,不然竟然会有返回值,不合理
```

```
;;由时间线索得到下一幕的游戏状态
(define (next-state-by-time state)
  (GS (GS-state state)
      (render-dinasor-by-time (GS-dinasor state))
      (render-ground-by-time (GS-ground state) (GS-speed state))
      (render-objects-by-time (GS-objects state) (GS-speed state))
      (render-score-by-time (GS-score state) (GS-speed state))
      (render-speed-by-time (GS-speed state))))
```

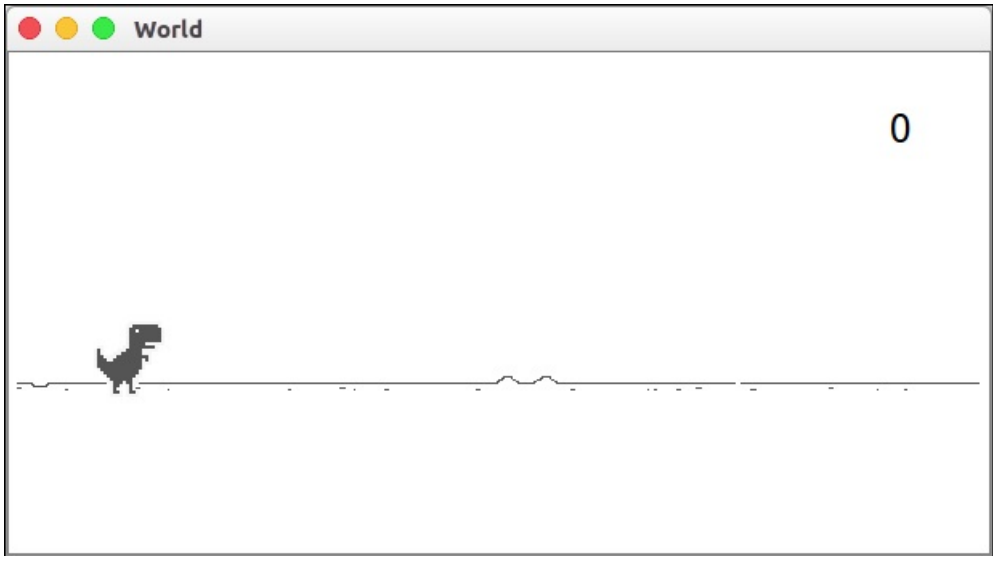
- 按键处理函数

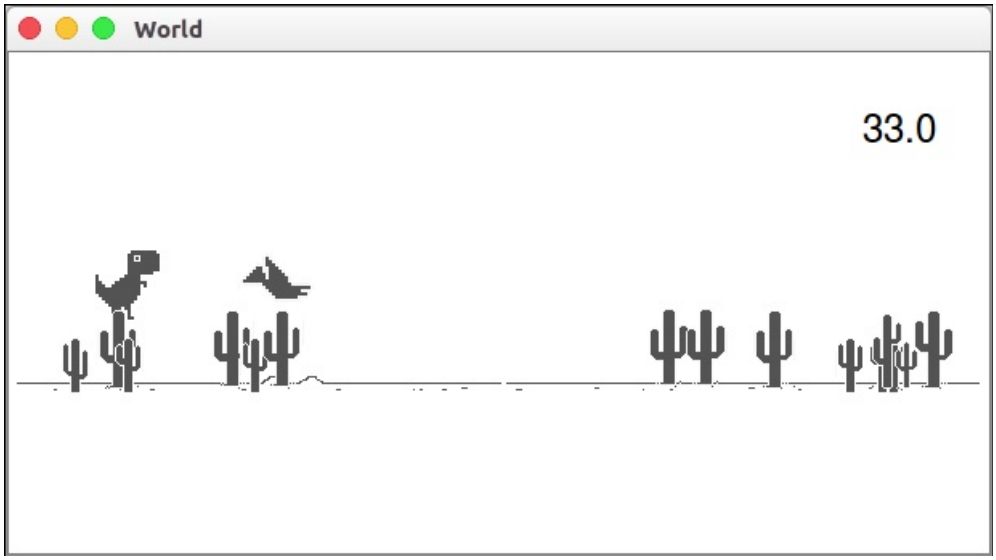
```
;;响应键盘事件
;;state key
;;==>new state
(define (key-render state a-key)
  (cond
    ;;判断当前的游戏的状态,如果不是running则不改变,从waiting or running ==>jump
    [(can-jump? state a-key) (letjump state)]
    [(can-down? state a-key) (letdown state)];;是否可以下蹲,可以则下蹲,从running变成d
    [(can-running? state a-key) (letrun state)];;从down变为running
    [(tiaoshi? a-key) (and(show-state state) state)]
    [(key=? "left" a-key) (speed-down state)]
    [(key=? "right"a-key) (speed-up state)]
    [else state];;其他情况则不变
    ))
```

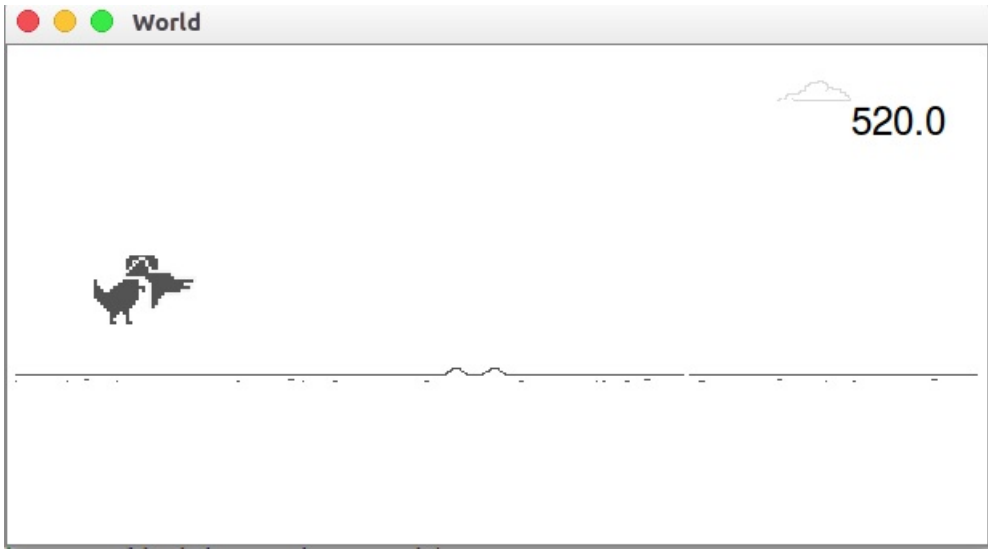
- 碰撞检测的代码已经在关键代码中说明

2 系统使用说明

2.1 游戏截图







2.2 操作说明

游戏有三种状态,waiting ,running,game over.

游戏操作者控制的恐龙有以下状态waiting,running,jump,down

在游戏进行的过程中恐龙的前进速度会越来越快,随之难度也会越来越大.游戏的控制者需要控制恐龙的行动让它取得越高的分数越好.

按键操作:上/space　　waiting -› running -›jump

下　　running -› down

左右　　调节相应的恐龙的前进速度

t　　输出游戏的当前状态,方便进行调试

3 团队分工说明

本项目由沈小双独立制作完成,包括选题,素材制作,游戏制作,文档编写