廣東工業大學

综合拓展报告

基于 CocosCreator 的 2d 冒险游戏设计

学　　院　计算机学院

专　　业　信息安全

年级班别　21 级 2 班

学　　号　3121005223

学生姓名　卢绪焕

指导教师　李斯

2024 年 10 月

目录

1 设计背景

1.1cocos2dx 的快速发展

Cocos2d-x 是 Cocos2d 中最受欢迎的跨平台框架，它支持多平台开发，并提供了丰富的功能库，例如物理引擎、粒子系统、音效管理等，方便开发者高效地开发出流畅、视觉效果良好的 2D 游戏。借助 Cocos2d-x，开发者能够创建具有精确碰撞检测、动画效果以及简单游戏逻辑的 2D 游戏。Cocos2d 还拥有广泛的社区支持，提供了丰富的学习资源和插件库，使其成为小型开发团队和独立开发者实现创意的理想选择。

1.2 设计内容

本报告则是基于 ts 语言，使用 Cocos2d 引擎开发的益智类冒险闯关游戏。玩家通过控制人物，击败敌人通关。

2 开发环境

2.1 编程语言 TypeScript

Cocos Creator 从 1.6 版本（发布于 2017 年 8 月）开始正式支持 TypeScript。在此之前，Cocos Creator 主要使用 JavaScript 进行开发。TypeScript 是 JavaScript 的超集，增加了静态类型检查，使得代码更易维护和调试。Cocos Creator 引擎层已经对 TypeScript 做了很好的支持，允许开发者使用 TS 来编写模块化、结构化的游戏代码。使用 TypeScript 开发 Cocos2d 游戏，可以在不牺牲运行效率的情况下提升开发体验。TypeScript 的类型检查和自动补全功能能够有效减少代码错误，使得游戏开发过程更高效、结构更清晰。

2.2 CocosCreator3.8.2

Cocos Creator 3.x 版本相较于之前的 2.x 版本进行了大幅度的提升和改进，尤其是在 3D

游戏开发、性能优化和多平台支持方面。尽管主要是为加强 3D 功能而推出的，但对 2D 游戏的开发也带来了显著的提升。3.x 中优化了渲染管线，支持多线程渲染和图形 API，提升了 2D 游戏的渲染性能。新的 UI 组件使得开发更复杂的 2D 界面更加容易。3.x 支持灵活的布局系统、自动调整分辨率等功能，使得 UI 在不同设备上的适配更容易。

3 需求分析

3.1 游戏规则

在游戏中，玩家会生成到第一关的起始位置，玩家需要控制人物进行前后左右四方向移动，通过顺时针旋转和逆时针旋转改变方向。当敌人在人物的正前方的时候，人物就会攻击敌人，使敌人死亡。而当人物在敌人的上下左右四个方向的时候，就会触发敌人的攻击，使玩家死亡。同时游戏中还应当存在地刺陷阱，玩家需要控制人物在合适的时机通过地刺，若被地刺攻击到，则玩家死亡。在击败一个关卡中的所有敌人之后，就能通过木门进入下一个关卡。

游戏还应设计三个按键，分别实现游戏的重新开始，回到玩家死亡前状态，退出按键。

3.2 角色设计

对人物设计应当包含如下的需求：
- 前后左右四个方向的移动，播放相应动画
- 顺时针旋转和逆时针旋转，播放相应动画
- 人物包含武器单位，当敌人在人物前方，且进入攻击距离时，攻击敌人。播放攻击动画
- 当人物执行完动作，或者原地站立时，播放站立动画
- 在移动中碰撞到瓦片地图墙体、旋转碰到墙体、移动出地图的时候都会播放碰撞动画，且无法继续移动
- 人物死亡后，播放死亡动画，且不能再移动。玩家可以选择回档或者重新开始游戏。

对敌人设计：
- 敌人不设计移动需求，仅循环播放站立动画
- 当人物进入前后左右四个方向一个单位的范围内时，攻击人物
- 死亡后播放死亡动画

4 架构设计

4.1 发布订阅模式（EventManager 事件中心）

在 Cocos 游戏开发中，发布订阅模式被广泛用于处理事件管理和模块通信。Cocos Creator 自带的 EventTarget 类是实现发布订阅模式的核心工具。通过发布订阅模式，开发者可以将游戏中的不同模块解耦，使得事件传递更灵活、更高效。

在实际应用中，其可以监听特定事件（如玩家按下按键使人物移动状态变化、人物攻击敌人、敌人攻击玩家、关卡完成等），在事件触发时自动更新内容。

4.2 Finite State Machine（FSM）有限状态机

状态机（State Machine）是一种用于管理系统在不同状态之间的转换。被广泛应用于处理有限状态的系统，以确保逻辑清晰、易于扩展和维护。

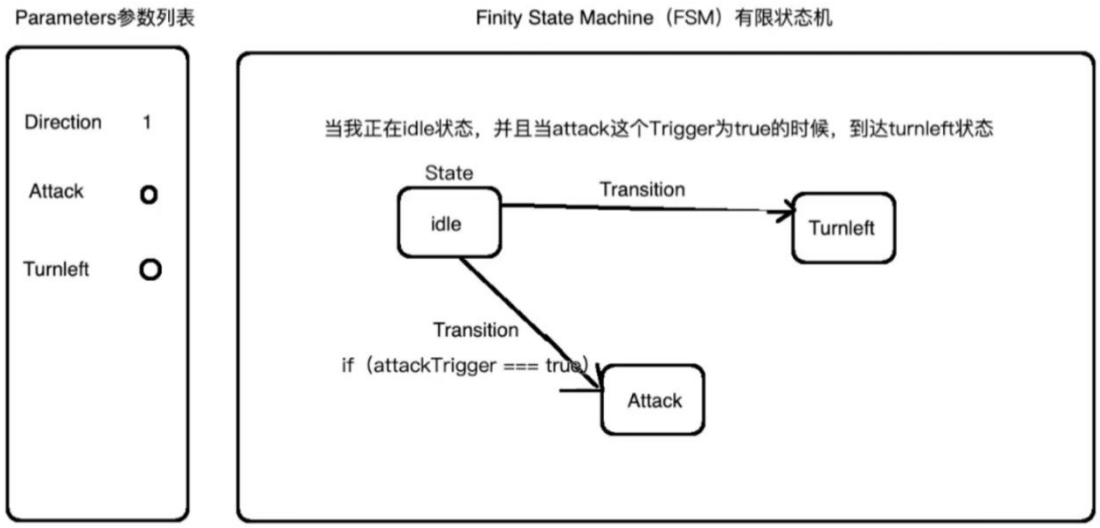


图 1 FSM 状态转换方式

Cocos Creator 3.4 引入了一个全新的 Marionette 动画系统，通过状态机控制对象的骨骼动画，实现了自动化、可复用的动画流程。在编辑器中，可以使用可视化工具的方式进行状态切换。但是动画状态非常的多，每个状态（idle、attack、turnleft、turnright）又包含了 top、bottom、left、right 四个方向的子状态，所以使用程序化编程的方式更为便捷。

4.3 数据和 UI 分离

数据和 UI 分离的思想是一种重要的设计架构，它常用于构建灵活、可维护、易扩展的程序。数据逻辑和界面展示彼此独立，数据和 UI 的更新独立进行，修改一方不会直接影响另一方。在前端开发中，数据和 UI 分离通常通过数据绑定或事件机制来实现。UI 组件监听数据变化或接收事件更新界面。而在本游戏的中也类似，例如控制人物进行移动的 move 方法，在玩家点击按钮进行移动操作的时候，不直接在 move 方法里改变 fsm，而是修改数据，通过数据驱动 fsm 状态机修改状态，从而实现数据分离。

```
//这段开始 UI 和数据分离
set state(newState){
this._state = newState;
this.fsm.setParams(this._state, true)
}
```

5 主要功能的具体实现

5.1 瓦片地图生成

这个类的主要任务是加载瓦片贴图资源，根据地图信息初始化每个瓦片，并将瓦片添加到当前节点中。 mapInfo 是从 DataManager 中获取的地图数据，包含地图的瓦片布局。tileInfo 是一个二维数组，用于存储初始化的瓦片对象。外层循环遍历每一行瓦片，内层循环遍历每一列瓦片。每个瓦片的数据保存在 mapInfo 的二维数组中。 item 是当前瓦片的数据，包括 src 和 type 属性。如果瓦片的 src 或 type 为 null，则跳过该瓦片。然后根据一定条件，将特定的瓦片样式随机化，给地图增加变化。最后，将创建的瓦片节点添加到 TileMapManager 的父节点中，实现瓦片在地图中的显示。

```
@ccclass('TileMapManager')
export class TileMapManager extends Component {
async init(){
const spriteFrames = await ResourceManager.Instance.loadDir("texture/tile/tile");
const {mapInfo} = DataManager.Instance;
DataManager.Instance.tileInfo = [];
for(let i = 0; i < mapInfo.length; i++){
const column = mapInfo[i];
DataManager.Instance.tileInfo[i] = [];
for(let j = 0; j < column.length; j++){
const item = column[j];
if(item.src === null || item.type === null){
continue;
}

let number = item.src;
if((number === 1 || number === 5 || number === 9) && i %2 == 0 && j % 2 == 0){
number += randomByRange(0,4);
}

const imgSrc = `tile (${number})`; //定义图片路径
const node = createUINode();
const spriteFrame = spriteFrames.find(v => v.name === imgSrc) || spriteFrames[5];
const tileManager = node.addComponent(TileManager);
const type = item.type
tileManager.init(type, spriteFrame,i, j);
DataManager.Instance.tileInfo[i][j] = tileManager;
```

```
node.setParent(this.node);
}
}
}
}
```

5.2 人物移动算法

输入方向判断：根据输入的方向 inputDirection 来决定角色的移动或转向。
移动逻辑：
向上移动 (CONTROLLER_ENUM.TOP)：将 targetY 减少 1，表示角色在 Y 轴上向上移动，同时将 isMoving 设置为 true。
向下移动 (CONTROLLER_ENUM.BOTTOM)：将 targetY 增加 1，表示角色在 Y 轴上向下移动，isMoving 设置为 true。
向左移动 (CONTROLLER_ENUM.LEFT)：将 targetX 减少 1，表示角色在 X 轴上向左移动，isMoving 设置为 true。
向右移动 (CONTROLLER_ENUM.RIGHT)：将 targetX 增加 1，表示角色在 X 轴上向右移动，isMoving 设置为 true。
转向逻辑：

向左转 (CONTROLLER_ENUM.TURNLEFT)：根据当前方向 this.direction 逐步改变角色的方向，转向左侧，同时设置状态为 ENTITY_STATE_ENUM.TURNLEFT。转向后，发出事件 EVENT_ENUM.PLAYER_MOVE_END，表示角色移动结束。

向右转 (CONTROLLER_ENUM.TURNRIGHT)：同样根据当前方向改变角色的方向，转向右侧，同时设置状态为 ENTITY_STATE_ENUM.TURNRIGHT，并发出 EVENT_ENUM.PLAYER_MOVE_END 事件。

```
move(inputDirection:CONTROLLER_ENUM){
if(inputDirection === CONTROLLER_ENUM.TOP){
this.targetY -= 1;
this.isMoving = true;

}else if(inputDirection === CONTROLLER_ENUM.BOTTOM){
this.targetY += 1;
this.isMoving = true;
```

```
}else if(inputDirection === CONTROLLER_ENUM.LEFT){
this.isMoving = true;
this.targetX -= 1;

}else if(inputDirection === CONTROLLER_ENUM.RIGHT){
this.isMoving = true;
this.targetX += 1;

}else if(inputDirection === CONTROLLER_ENUM.TURNLEFT){
if(this.direction === DIRECTION_ENUM.TOP){
this.direction = DIRECTION_ENUM.LEFT
}else if(this.direction === DIRECTION_ENUM.LEFT){
this.direction = DIRECTION_ENUM.BOTTOM;
}else if(this.direction === DIRECTION_ENUM.BOTTOM){
this.direction = DIRECTION_ENUM.RIGHT;
}else if(this.direction === DIRECTION_ENUM.RIGHT){
this.direction = DIRECTION_ENUM.TOP;
}
EventManager.Instance.emit(EVENT_ENUM.PLAYER_MOVE_END);
this.state = ENTITY_STATE_ENUM.TURNLEFT;
}else if(inputDirection === CONTROLLER_ENUM.TURNRIGHT){
if(this.direction === DIRECTION_ENUM.TOP){
this.direction = DIRECTION_ENUM.RIGHT
}else if(this.direction === DIRECTION_ENUM.LEFT){
this.direction = DIRECTION_ENUM.TOP;
}else if(this.direction === DIRECTION_ENUM.BOTTOM){
this.direction = DIRECTION_ENUM.LEFT;
}else if(this.direction === DIRECTION_ENUM.RIGHT){
this.direction = DIRECTION_ENUM.BOTTOM;
}
this.state = ENTITY_STATE_ENUM.TURNRIGHT;
EventManager.Instance.emit(EVENT_ENUM.PLAYER_MOVE_END);
}
}
```

5.3 动画加载

PlayerStateMachine 继承自 StateMachine，是一个玩家状态机的实现。它主要负责管理和控制玩家的各种状态切换，以及相应的动画效果。下面是其主要的几个函数:

initParams()初始化玩家状态机的所有参数，并将其存储在 this.params 中。调用 getInitParamsTrigger() 和 getInitParamsNumber() 方法为各个状态参数创建初始值。PARAMS_NAME_ENUM 列举了玩家的各种状态，如 IDLE、TURNLEFT、TURNRIGHT 等。每个参数表示一个状态的触发条件或数值，比如 DIRECTION 是数值类型的参数，IDLE 是触发器类型的参数。

```
initParams(){
this.params.set(PARAMS_NAME_ENUM.IDLE,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.DIRECTION, getInitParamsNumber());
this.params.set(PARAMS_NAME_ENUM.TURNLEFT,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.TURNRIGHT,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.BLOCKFRONT,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.BLOCKBACK,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.BLOCKLEFT,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.BLOCKRIGHT,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.BLOCKTURNLEFT,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.BLOCKTURNRIGHT,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.DEATH,getInitParamsTrigger());
this.params.set(PARAMS_NAME_ENUM.ATTACK,getInitParamsTrigger());
}
```

initStateMachine()初始化各个状态的子状态机并将其与特定的状态参数关联，存储在 this.stateMachines 中。例如，this.stateMachines.set(PARAMS_NAME_ENUM.IDLE, new IdleSubstateMachine(this)) 将 IDLE 状态映射到 IdleSubstateMachine。这些子状态机（如 IdleSubstateMachine, TurnLeftSubStateMachine）分别处理玩家的不同状态逻辑和动画，便于在运行时动态切换状态。

```
initStateMachine(){
this.stateMachines.set(PARAMS_NAME_ENUM.IDLE, new IdleSubstateMachine(this));
//
this.stateMachines.set(PARAMS_NAME_ENUM.TURNLEFT,                    new
TurnLeftSubStainMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.TURNRIGHT,                   new
TurnRightSubStainMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.BLOCKFRONT,                  new
BlockFrontSubstateMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.BLOCKBACK,                   new
```

```
BlockBackSubstateMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.BLOCKLEFT, new
BlockLeftSubStateMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.BLOCKRIGHT, new
BlockRightSubStateMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.BLOCKTURNLEFT, new
BlockTurnLeftSubStateMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.BLOCKTURNRIGHT, new
BlockTurnRightSubStateMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.DEATH, new DeadSubStateMachine(this));
this.stateMachines.set(PARAMS_NAME_ENUM.ATTACK, new AttackSubStateMachine(this));
}
```

initAnimationEvent()初始化动画事件，在动画播放完成时触发指定回调。注册了 Animation.EventType.FINISHED 事件监听器，当动画完成时，会检查 defaultClip.name 的名称，如果名称中包含 turn 或 block，会将玩家状态切换为 IDLE。通过这种方式，状态机可以在特定动画结束后自动返回到默认的 IDLE 状态。例如我们希望人物在攻击之后，继续播放站立动画，而不是静止。

```
initAnimationEvent(){
this.animationComponent.on(Animation.EventType.FINISHED,()=>{
const name = this.animationComponent.defaultClip.name;
const whiteList = ['turn', 'block', 'attack']; //v 名字里包含有 turn 和 block，之后都会让它回到 idle
状态
if(whiteList.some(v=>name.includes(v))){
this.node.getComponent(EnityManager).state = ENTITY_STATE_ENUM.IDLE;
}
})
}
```

run() 函数负责根据当前状态和触发条件来进行状态切换。它遍历不同的状态条件，检查 this.params 中的参数值来确定需要激活的状态。在满足特定条件时，会将 this.currentState 切换到对应的子状态机，如 BLOCKFRONT 状态将切换到 BlockFrontSubstateMachine。如果没有任何参数触发，则保持 currentState 不变。

```
run(){
switch(this.currentState){
case this.stateMachines.get(PARAMS_NAME_ENUM.IDLE):
case this.stateMachines.get(PARAMS_NAME_ENUM.TURNLEFT):
case this.stateMachines.get(PARAMS_NAME_ENUM.TURNRIGHT):
case this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKFRONT):
case this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKBACK):
case this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKLEFT):
```

```
case this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKRIGHT):
case this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKTURNLEFT):
case this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKTURNRIGHT):
case this.stateMachines.get(PARAMS_NAME_ENUM.DEATH):
case this.stateMachines.get(PARAMS_NAME_ENUM.ATTACK):
if(this.params.get(PARAMS_NAME_ENUM.BLOCKFRONT).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKFRONT)
}else if(this.params.get(PARAMS_NAME_ENUM.BLOCKBACK).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKBACK)
}else if(this.params.get(PARAMS_NAME_ENUM.BLOCKLEFT).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKLEFT)
}else if(this.params.get(PARAMS_NAME_ENUM.BLOCKRIGHT).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKRIGHT)
}else if(this.params.get(PARAMS_NAME_ENUM.BLOCKTURNLEFT).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKTURNLEFT)
}else if(this.params.get(PARAMS_NAME_ENUM.BLOCKTURNRIGHT).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.BLOCKTURNRIGHT)
}else if(this.params.get(PARAMS_NAME_ENUM.TURNLEFT).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.TURNLEFT)
}else if(this.params.get(PARAMS_NAME_ENUM.TURNRIGHT).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.TURNRIGHT)
}else if(this.params.get(PARAMS_NAME_ENUM.IDLE).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.IDLE)
}else if(this.params.get(PARAMS_NAME_ENUM.DEATH).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.DEATH)
}else if(this.params.get(PARAMS_NAME_ENUM.ATTACK).value){
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.ATTACK)
}else{
this.currentState = this.currentState;
}
break;

default:
this.currentState = this.stateMachines.get(PARAMS_NAME_ENUM.IDLE)
}
}
```

所有的 SubStateMachine 都类似，只是文件的路径不一样，这里只展示 AttackSubStateMachine。根据不同的攻击方向设置对应的 State 实例，通过 stateMachines 管理攻击动画的四个方向状态。在实际使用中，状态机会根据玩家的攻击方向切换到相应的动画资源，达到根据攻击方向播放不同动画的效果。

```
const BASE_URL = 'texture/player/attack'

export default class AttackSubStateMachine extends DirectionSubStateMachine{
constructor(fsm: StateMachine){
super(fsm)
this.stateMachines.set(
DIRECTION_ENUM.TOP,
new State(fsm, `${BASE_URL}/top`))

this.stateMachines.set(
DIRECTION_ENUM.BOTTOM,
new State(fsm, `${BASE_URL}/bottom`))

this.stateMachines.set(
DIRECTION_ENUM.LEFT,
new State(fsm, `${BASE_URL}/left`))

this.stateMachines.set(
DIRECTION_ENUM.RIGHT,
new State(fsm, `${BASE_URL}/right`))
}
}
```

6 测试

6.1 四方向移动和碰撞阻塞

图 2 向上移动

图 3 向下移动

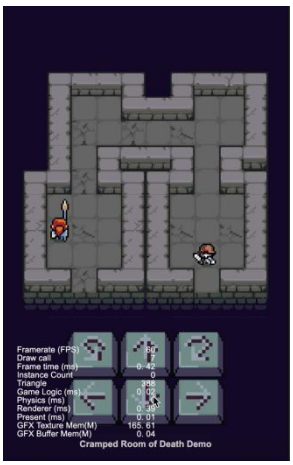


图 4 向左移动

图 5 向右移动

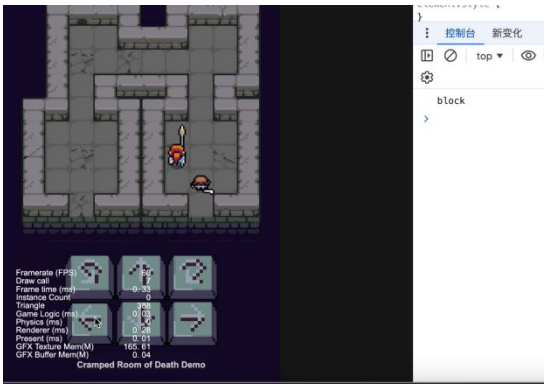


图 6 左 block

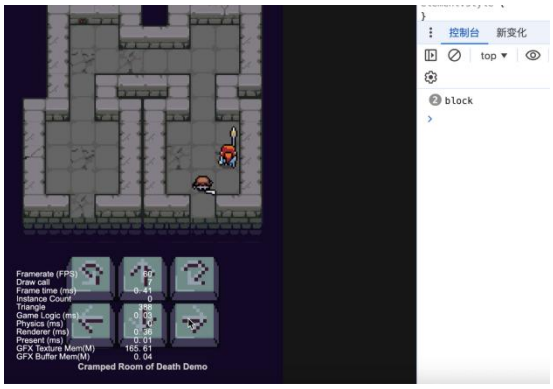


图 7 右 block

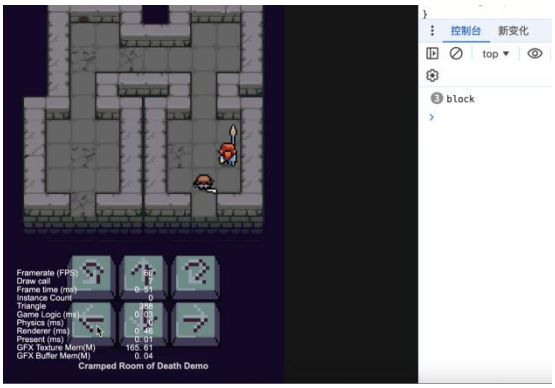


图 8 上 block

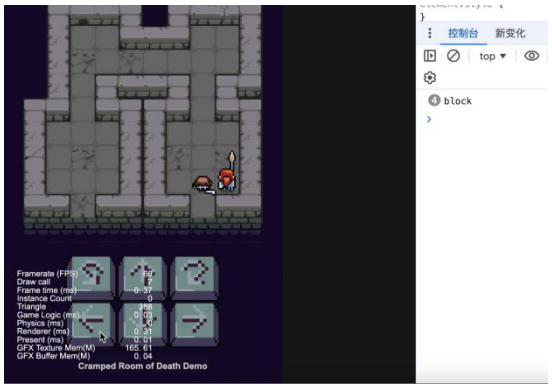


图 9 下 block

6.2 左右旋转和碰撞阻塞

图 10 右旋转

图 11 左旋转

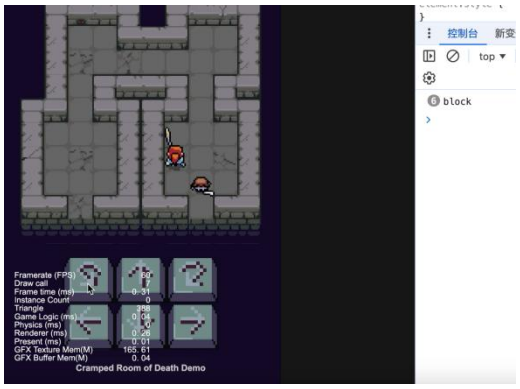


图 12 左旋转 block

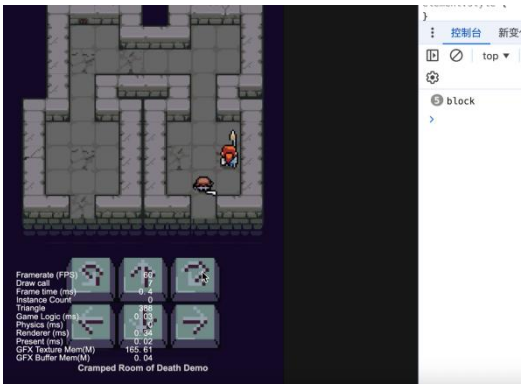
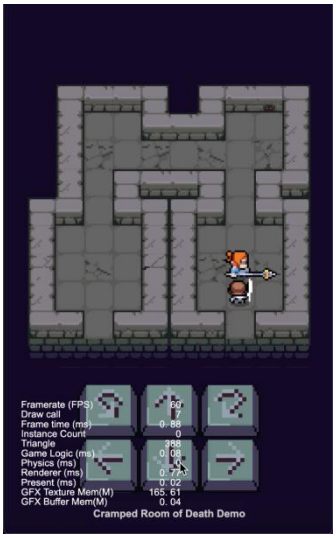


图 13 右旋转 block

6.3 人物和怪物攻击

图 14 人物攻击图



15 怪物攻击

6.4 人物和怪物死亡

图 16 怪物死亡图

17 人物死亡

6.5 进入下一关

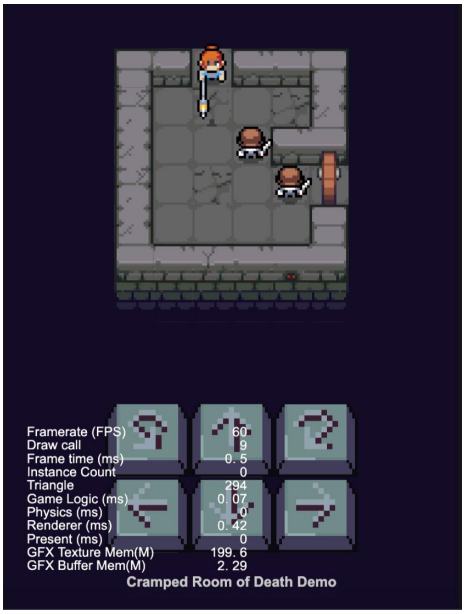


图 18 进入下一关

7 项目结构

7.1 抽象基类

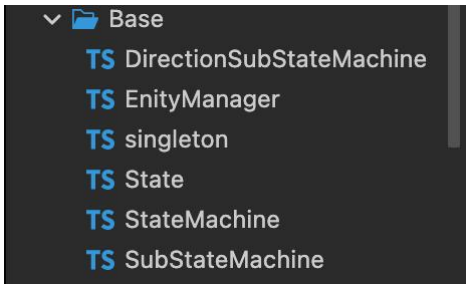


图 19 抽象基类

7.2 枚举和地图信息

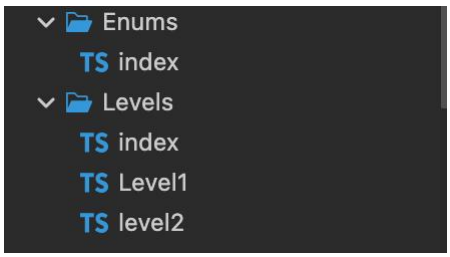


图 20 枚举和地图信息

7.3 数据、资源、事件中心

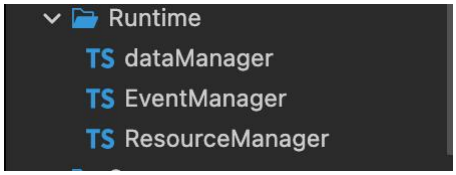


图 21 数据、资源、事件中心

7.4 玩家控制中心及其子状态机

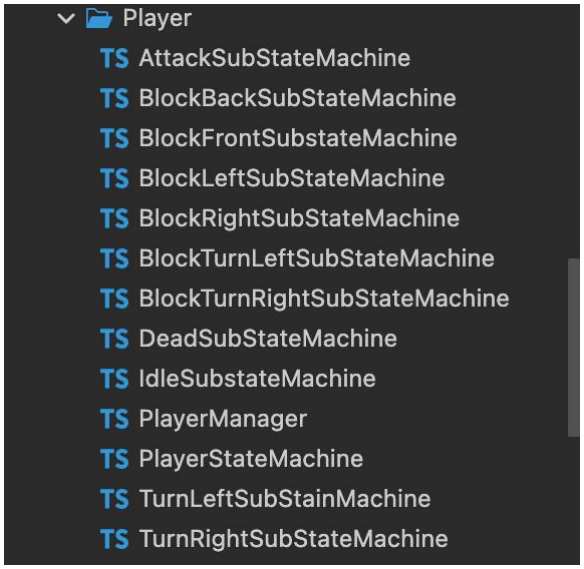


图 22 玩家控制模块和子状态机

7.5 场景、瓦片、UI 控制中心

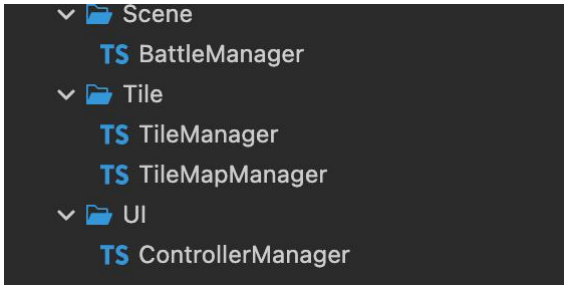


图 23 场景、瓦片、UI 控制中心

7.6 怪物控制中心及其子状态机

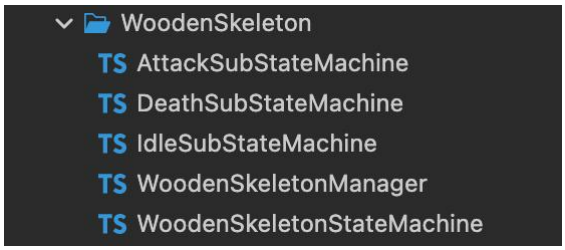


图 24 怪物控制中心及其子状态机

8 感想

做一个游戏是我一直以来想做但是又没有能做的事情。之前有机会参与过 cocos 引擎官方办的线下沙龙活动，借此专项设计的契机，我选择了使用 cocos 做一个 2d 游戏的想法。但是由于我不是非常熟悉 ts 和游戏开发，一切都是刚刚开始，我遇到了非常多且繁琐的困难。于是我只能不断的复制报错到 csdn、谷歌和 github 等平台上寻求我想要的答案。再后来，我学会了查阅官方文档，取得了不少的进展。令我影响最深刻的是我第一次在 cocos 的论坛上发帖寻求帮助，我把我的需求、报错信息、代码和截图都粘贴了上去，等待有人能解答我的问题。令人欣喜的是，我在第二天帖子审核通过之后，一两小时后就有大佬给出他的解决方法，我成功的复现，解决问题的时候心情是十分欢喜的。

通过这次实践，我掌握了一定的 cocos 开发经验和 ts 基础。从 0 开始慢慢摸索，通过各种办法去学习它，然后再到初步了解和掌握一项新技能的这个过程，我想它对我以后的成长大有裨益。遗憾的是目前没有办法再投入更多的精力继续完善它了，或许等我再有时间，我会继续打开它，完成它的。