科研和工程中的C++编程 课程设计总报告

作者	学号
陶新野	3160103871
唐小虎	3160103866
袁谱博	3160105215

项目总体框架

游戏实现功能描述

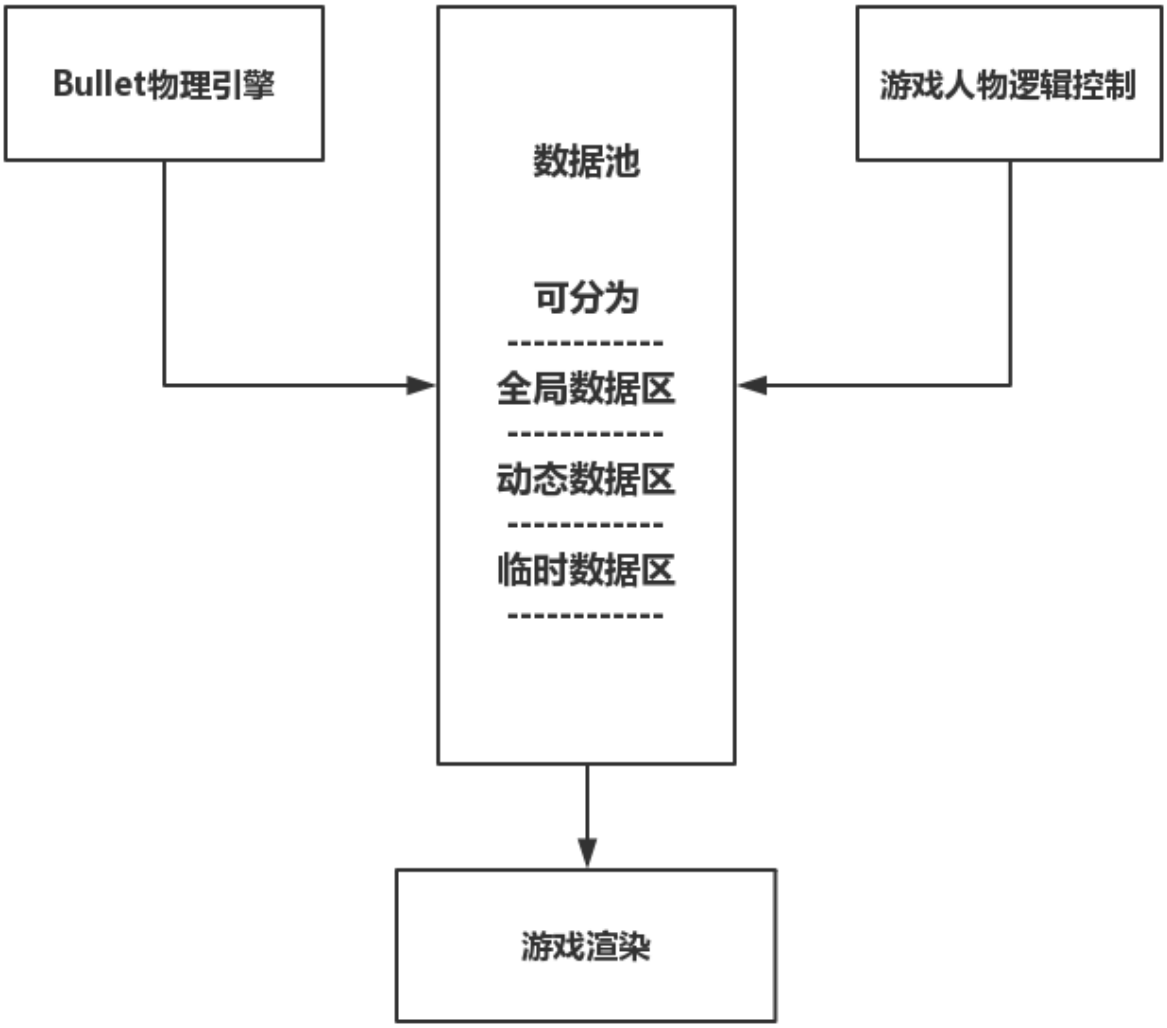
本项目Happy Bird 是一款基于OpenGL图形库和Bullet3 物理引擎的3D游戏。在游戏中我们实现了3D画面，游戏音效，积分系统，角色控制，以及一套完整的游戏逻辑。玩家在游戏中可通过键盘鼠标控制主人公角色，在紧张刺激的关卡中与怪兽斗智斗勇，体验成为一名合格冒险者的乐趣!

游戏系统设计结构

整体架构

游戏整体设计架构可分为渲染端、物理引擎端、逻辑控制端，三方共享数据并实现以数据为路径的通讯模式。

详细框架如图:



游戏包括以下主要元素

游戏角色

- 主人公 主人公外观由 2D贴图生成，在游戏中通过第三人称视角控制。玩家通过 W-A-S-D，空格(Space) 按键可控制主人公在游戏空间中实现三维移动。按键FG分别对应两种攻击模式，主人公角色有一定血量，且在进行攻击过程中可触发血量增加功能。
- 怪兽 怪兽外观的生成原理与主人公角色类似。其活动方式为当距离主人公角色较远时随机游走，当怪兽与主人公角色距离小于设定阈值时，怪兽主动向主人公角色靠近并发起攻击。在游戏中，怪兽的位置与速度为一定范围内的随机变量，当其血量小于0时即消亡。

游戏场景与道具

- 石头地形 地形在特定参数控制下随机生成，且带有3D效果和地形边界。
- 木箱 箱子在游戏不具备攻击能力，自身亦不会主动运动，但是可以在外力作用下实现碰撞与滑动。

- 重力场 我们在游戏实现了重力作用效果，游戏中的物体如主人公角色，怪兽，木箱，乃至攻击粒子均都受到重力作用。

游戏视效

- 视角切换 游戏中的3D视角切换通过鼠标移动实现，玩家通过移动鼠标可以自由转换观察的角度与方向。
- 光照 我们对石砖墙壁/木箱使用了不同的着色器，使其在画面中呈现出逼真的材质效果。且在玩家移动过程中，光照会随着主人公角色移动而发生变化。
- 攻击 为玩家设计了两种攻击模式：大范围枪械攻击和高精准激光攻击，并有相应的视觉特效。
  火焰/烟雾/血槽 在受到攻击后，视具体角色而产生相应的伤害，并以飘出减号和加号来直观表达。

游戏实现流程及具体工作

设计流程

核心逻辑设计

第一阶段为设计游戏的核心逻辑，即以主人公角色的血量为主线，以设计动作和怪兽攻击作为影响主人公血量的因素。游戏中的其它元素均围绕此主线展开。

三维场景设计

在3D场景设计上，负责场景设计的同学为游戏布置了一个有界的游戏空间，游戏空间分为高低两层，二者用石梯加以连接，同时还增添了可移动的箱子。

集成计分系统与其它游戏元素

在游戏设计的第3阶段，我们增加了积分系统，每当玩家击中或消灭怪兽，即可获得相应的积分奖励。

日程计划

团队采用敏捷开发快速迭代的开发方式，每完成一个单元模块并通过测试后都会发布一个新版本。

具体操作

w 向z轴负方向移动 a 向 x轴负方向移动 s 向 z轴正方向移动 d向 x轴正方向移动 space 向y轴正方向跳跃 f 发起普通攻击 g 发起粒子攻击

游戏实现分工

陶新野：整体架构设计，物体抽象层及缓存池构建，物体操控接口设计，特效渲染

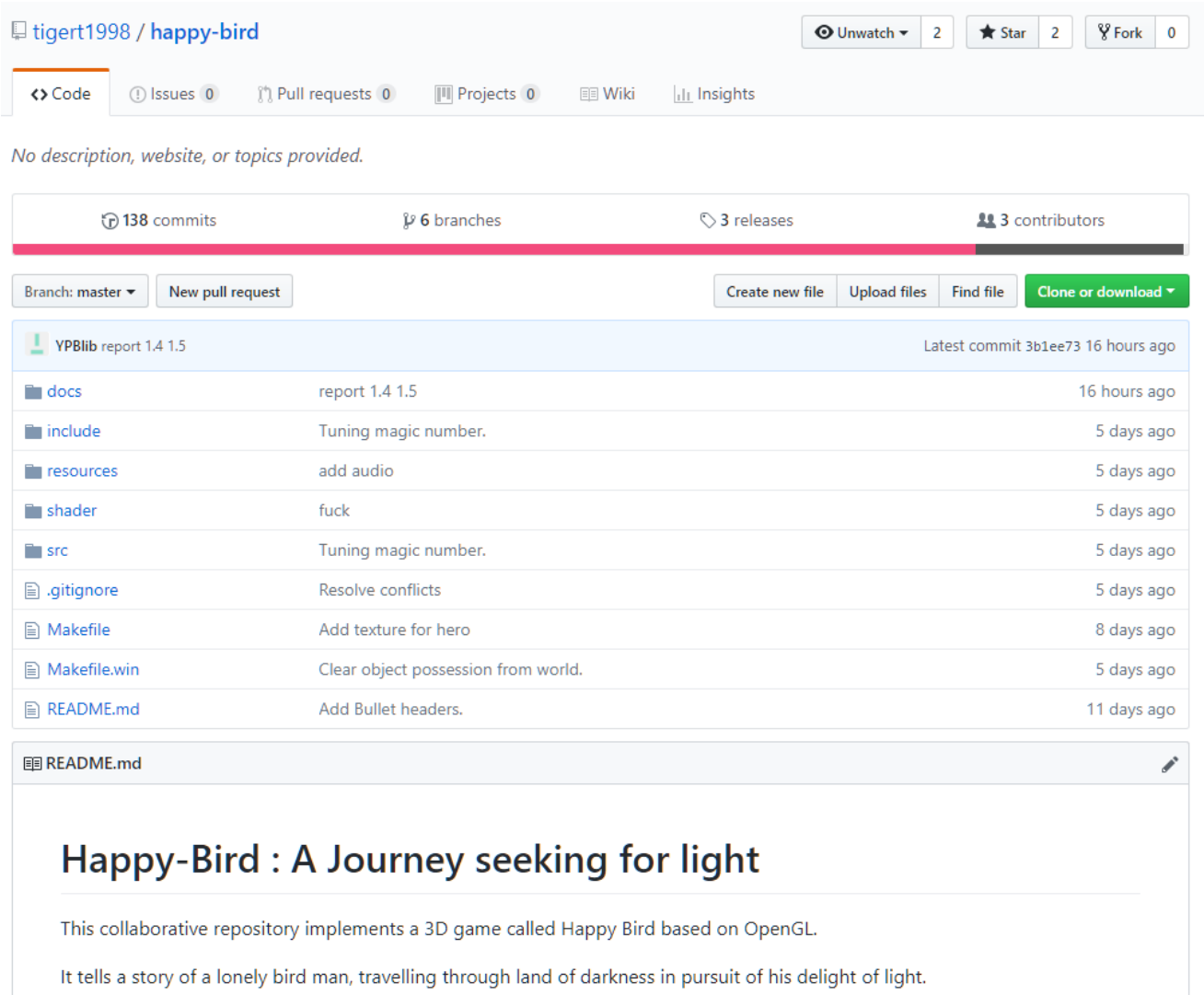
唐小虎：整体架构设计，渲染基础设施编写，人物控制自动机设计，玩家内存池

袁谱博：产品规划设计， 游戏音效模块设计，bullet物理引擎接口整合

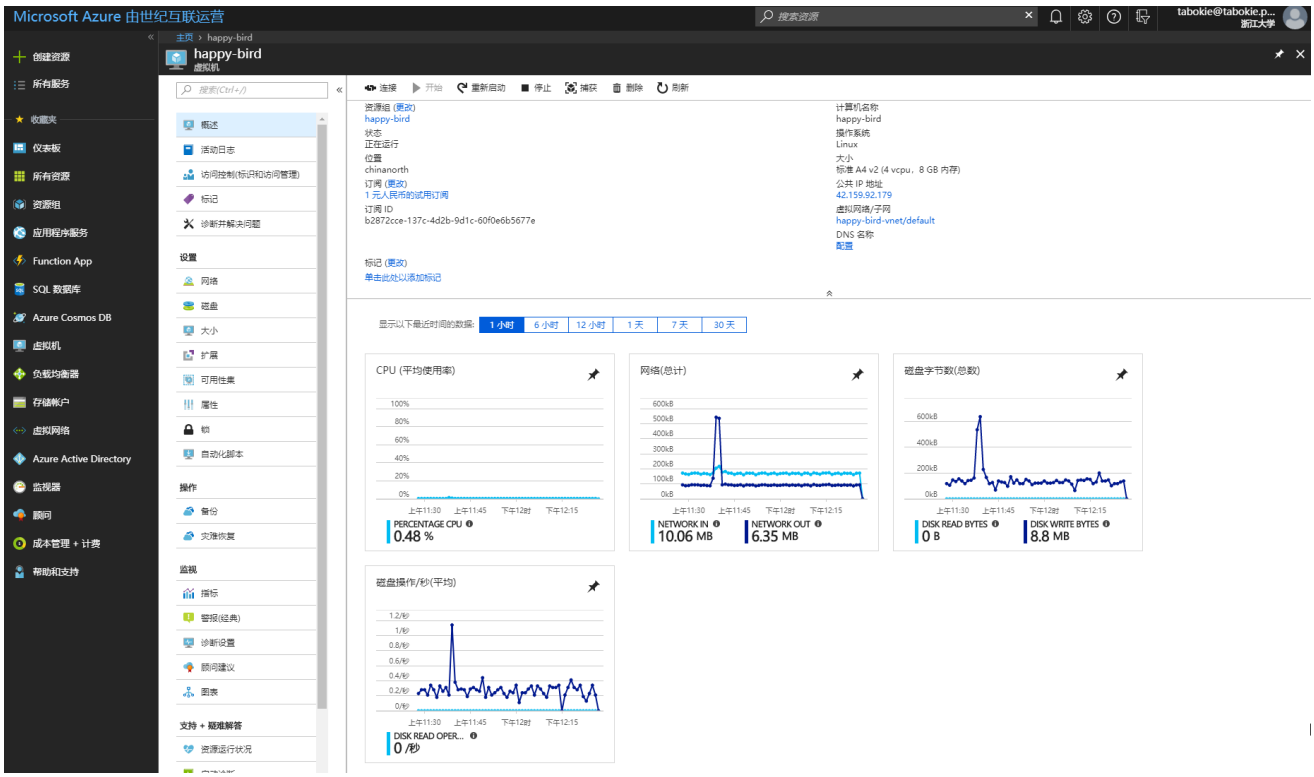
项目协作环境和参考配置

项目协作环境

- 版本控制与协作开发：我们在GitHub网站上建立代码仓库，以便进行团队分工协作，每个团队成员根据自身任务开设新分支，且只修改、提交自己所负责部分的代码，进而实现团队分工中的解耦和，彼此任务进度互不影响，极大地提高了开发效率，



- 在持续集成 (Continuous integration) 方面，团队成员在自己的服务器上搭建了Jenkins服务，并将其用于进度监视与审查。



编程语言

本工程全部使用C++语言编写，最低支持标准C++11。

开发环境

OpenGL

Bullet

对于想要构建、开发本项目的用户，您可考虑以下工具链：

MinGW (Minimalist GNU for Windows) CMAKE

此外，我们在开发过程中还使用以下代码规范检查与单元测试工具：

code review:

- cpplint
- cppcheck
- cppncss

unit test:

- RenderDoc
- GoogleTest

各模块设计介绍

Bullet物理引擎连接

主要功能

Bullet物理引擎是一个开源的第三方库，提供了较为完备的物理模拟解算支持。其中包含了各类刚体、织物、软体等丰富的物理建模框架。其实时解算能力也足以完成本次游戏的实时计算量要求。

然而Bullet引擎是建立在纯数学解算模型上的物理引擎，不具备针对计算机图形显示的优化和数据直连。为了将Bullet引擎与前端OpenGL渲染相连接，我们需要手动由抽象的数学模型实体化和参数化为实际的三角形面片信息。

接口设计

Bullet物理引擎中的数据和解算工作主要由`btCollisionWorld`承担，`btCollisionWorld`下又派生`btDynamicWorld`，以及本次项目中使用的`btRigidSoftWorld`，用于实现软体和刚体兼容的物理模拟。

在下一抽象层级，Bullet物理引擎提供了`btCollisionShape`和`btRigidBody/btSoftBody`来作为物理实体的容纳容器。其下设丰富的接口函数来实现游戏中的实时操控。

GLSL着色器封装类

主要功能

为了在程序设计中方便的管理OpenGl着色语言GLSL相关的工作，我们设计了着色器封装Shader类，通过这一封装我们能在程序中轻松设置着色过程中的环境变量，以及更改着色风格。并且封装具备完备的异常处理机制，便于处理渲染相关的错误问题。

接口设计

```
class Shader {
public:
    Shader() = delete;
    Shader(const std::string &vs_path, const std::string &fs_path);
    void Use() const;
    template <typename T> void SetUniform(const std::string &identifier,
const T&) const;

private:
    const FileManager file_manager = FileManager();

    static uint32_t Compile(GLenum type, const std::string &source,
const std::string &path);

    static uint32_t Link(uint32_t vs_id, uint32_t fs_id);
```

```
    uint32_t id;
};
```

其中默认构造函数被显式删除了，只提供`Shader::Shader(const string &, const string &)`的构造函数，两个字符串分别是定点着色器和像素着色器的地址；`Shader::Use()`只是`glUseProgram(uint32_t)`的封装。

抽象物体封装 Object

主要功能

物体抽象主要提供了共用基类接口设计，作为游戏全局所有可绘制实体的父类接口，它必须具有绘制和绘制更新的方法声明，同时，为了实现物体受控运动的游戏逻辑，还需要提供一系列运动参数的set方法，另外，为了在全局操作中获得物体位置，并提供每个物体和被其他物体附着的属性，Object类还需提供详尽的get方法。

在此基类的派生类中，我们提供了丰富的建模元素，从基本的刚体如球体、长方体，到复杂的软体如织物，以及抽象物体如粒子。这些建模元素与光照和贴图组合构成了我们游戏世界中的砖瓦人物。

接口设计

```
// 绘制及更新方法
virtual void Update(float);
virtual void Draw(const Camera&, const LightCollection&);
// Get accessors
virtual btVector3 GetOrigin(void);
virtual btTransform GetTransform(void);
virtual btVector3 GetVelocity(void);
// Set accessors
virtual void SetVelocity(const btVector3&);
// 其他共用接口
virtual void Attach(std::weak_ptr<Object>);
```

物体数据集合 Collection

主要功能

在底层独立的Object之上，我们使用了Collection类作为数据内存池保管这些动态变化的数据。

对应与高层架构中的三个部分，分别设计了三个Collection池：StageCollection、PlayerCollection、TempCollection。它们分别具有全局不可变、动态可变、动态易失的特性。为了更好地处理期间的内存管理，我们大量使用了智能指针如shared_ptr和weak_ptr来进行数据创建和引用传递。

接口设计

此处讨论一般性的Collection类接口设计，为了方便外端对内存池内部数据进行遍历，我们提出两种方案，并都在实际编码中投入使用。它们分别是迭代器返回和函数式传递。

对于迭代器返回，我们提供接口`objectBegin(void)`，其返回一个自定义结构体的原生迭代器，同时要求此结构体继承于全局的可迭代基类，此基类规定了所有返回的迭代类型必须具备接口`get()`，而get方法则是由此结构返回weak_ptr供外层安全引用。

游戏人物控制接口 Character

主要功能

Character类作为一个接口类，保有指向Object物体的成员引用，并提供了更为易用的控制接口封装。在当前的游戏设计中，这一封装包含了基本运动如前进后退、跳跃，以及复杂行为如攻击、交谈。这些简单的命令可以看作全局逻辑向后端数据发出的命令，作为对响应命令的响应，人物的物理状态会发生改变，甚至创建新物体。

接口设计

接口设计如下：

```
void Move(bool, float);
void Rotate(bool, float);
void Jump(float);
void LaserAttack(void);
void BoxAttack(void);
```

其中布尔量标识移动方向，初步实现了两种攻击方式。

每一种可量化的移动方式都对应了输入参数的时间值，在程序中，使用每一帧刷新间的绝对时间差作为这一参数的输入，使得人物的运动状态不随计算效率和帧率的变化发生大的偏移。同时，为了避免键盘时间短时间大量触发导致人物运动不自然，我们设置了运动速度上限，这一上限与人物的内部数值配置有关，同时，每一帧停止运动响应后对响应人物进行reset操作，清除冗余的速度信息，使得人物运动符合玩家预期的同时也遵循物理引擎的计算规则。

游戏人物控制中心 Controller

主要功能

Controller作为整体，具有执行者的功能和地位，直接执行对可操控物体的控制。具体来说，我们特化实现了两种`Controller`：`KeyboardController`和`AutomationController`，它们分别对应了玩家控制和NPC控制。玩家控制通过键盘事件触发，并由Controller转接至实际的人物角色进行执行。而NPC的控制则完全由计算机程序执行，根据自身预定义行为模式和当前环境而产生一个自动状态机，在此状态机的操控下其他人物得以正常的运作行为。

接口设计

```
class Controller {
protected:
    Character &controlee_;

public:
    Controller() = delete;
    Controller(Character &controlee);
    virtual void Elapse(double time) = 0;
};
```

其中`Character`是一个拥有`Move`，`Rotate`或是`Jump`等操作方式的角色（这样的角色可以是人类角色、可以是怪物等等）。而`Controller`的作用就是抽象出它们的运动逻辑并操控这一类`Character`。

另外`Character::Elapse(double)`的作用是刷新角色模拟。因为真正连续的操作是不可能实现的，可以实现的只有在一帧一帧画面之间离散的模拟运动逻辑。它被设计成一个纯虚函数，留给子类单独实现。

测试方案和运行结果

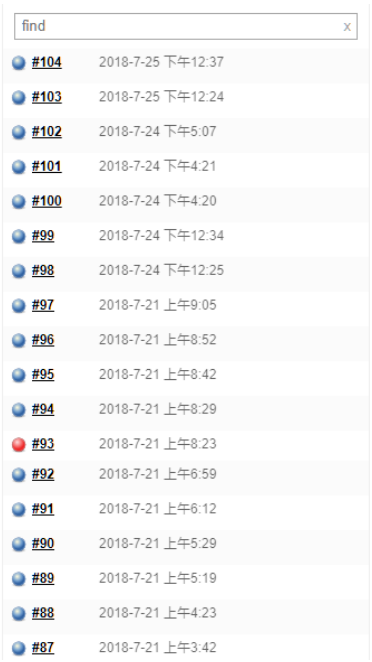
测试框架和部署

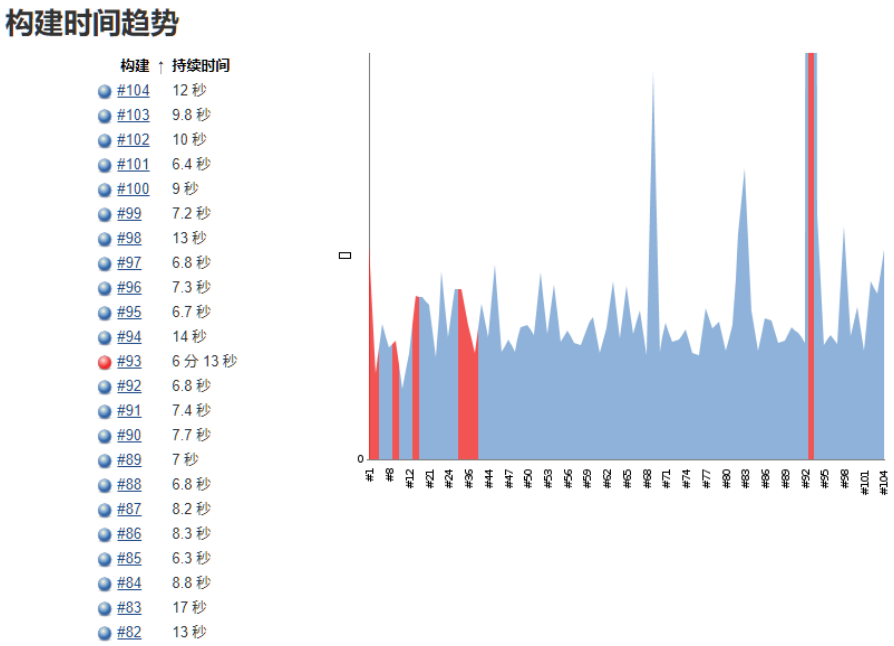
如前所述，这个团队项目使用了Jenkins持续继承平台和单元测试工具Google Test，我们的单元测试文件便全部使用Google Test编写部署，并设置Jenkins为对Commit响应，每一次Commit都执行编译和单元测试运行的流程，将单元测试结果反馈至Jenkins平台。

实际操作过程中的使用截图：

平台主页：

测试结果汇总和统计：





物理引擎测试

由于使用了第三方开源物理引擎Bullet，很难在短时间内对这一庞大复杂系统进行细致的测试，我们只简单地使用了几种基本场景对其进行测试。

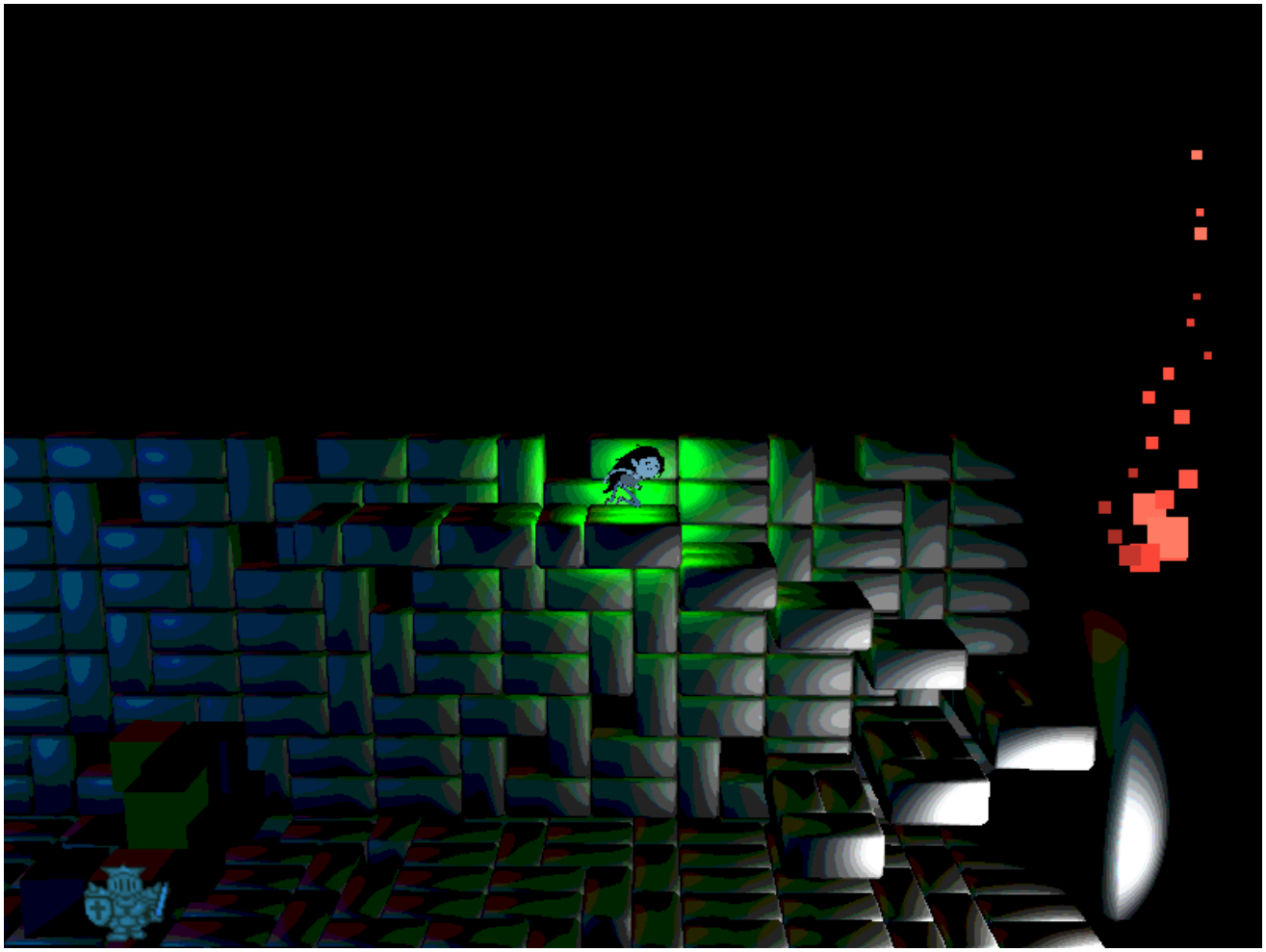
一个典型的测试样例如下：

```
TEST(BulletEngineTest, TestBox){
    bt_configure_ = new btSoftBodyRigidBodyCollisionConfiguration();
    bt_dispatcher_ = new btCollisionDispatcher(bt_configure_);
    bt_overlapping_paircache_ = new btDbvtBroadphase();
    bt_solver_ = new btSequentialImpulseConstraintSolver;
    bt_soft_solver_ = new btDefaultSoftBodySolver;
    bt_world_ = new btSoftRigidDynamicsWorld(
        bt_dispatcher_,
        bt_overlapping_paircache_,
        bt_solver_,
        bt_configure_,
        bt_soft_solver_
    );

    bt_world_->setGravity(btVector3(0, -50, 0));
    bt_soft_info_.m_dispatcher = bt_dispatcher_;
    bt_soft_info_.m_broadphase = bt_overlapping_paircache_;
    bt_soft_info_.m_sparsesdf.Initialize();

    bt_world_->addRigidBody(new
btRigidBody(btRigidBodyConstructionInfo(0, nullptr, btBox)));
    ASSERT_TRUE(bt_world_->getRigidBodyArray != nullptr);
    EXPECT_EQ(bt_world_->getRigidBodyArray[0].getOrigin(), btVector3(0,
0, 0));
}
```

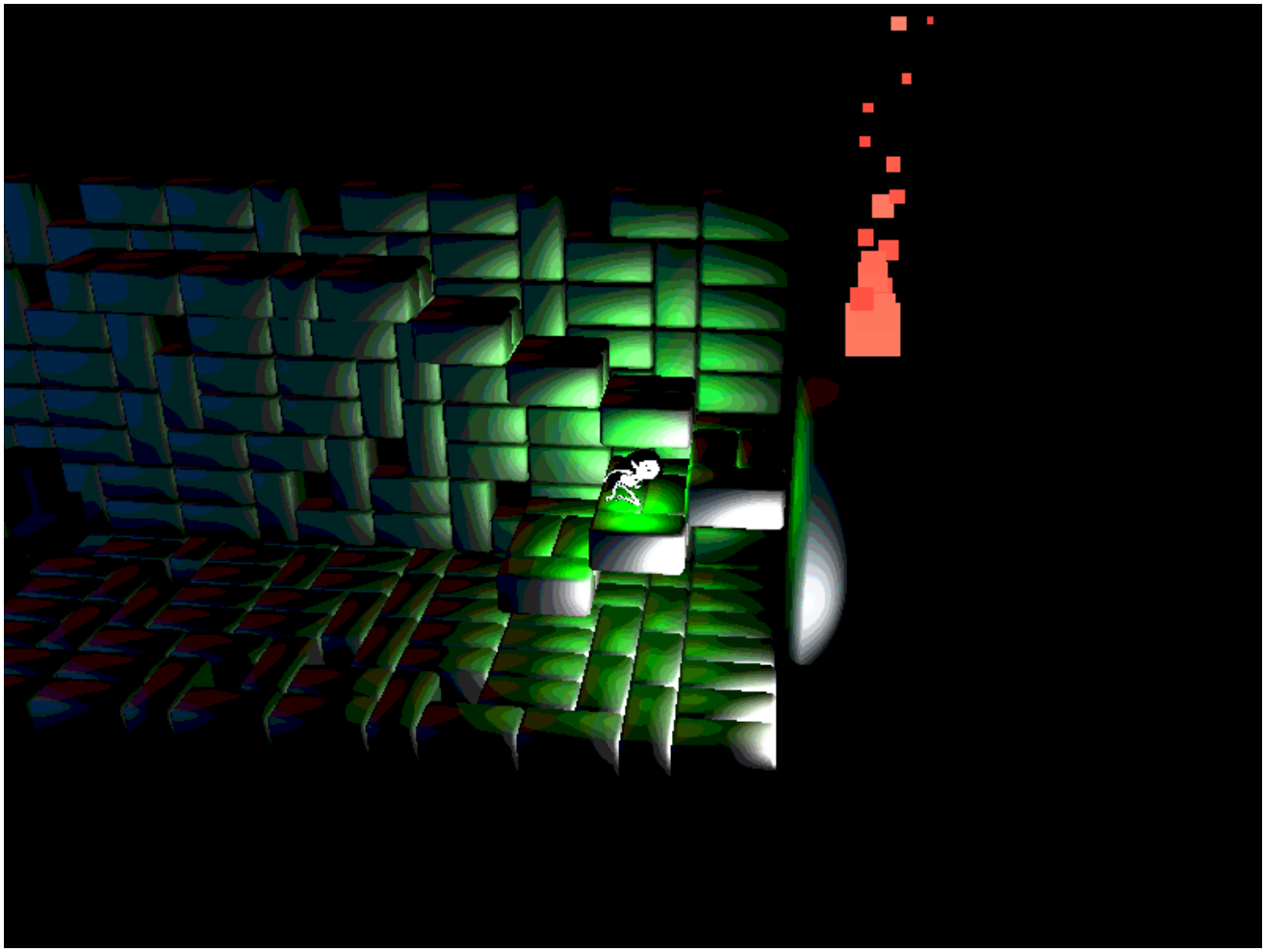
事实上在实际使用中也出现了一些原因不明的内存泄漏或是断言错误问题，对这些可能由于第三方测试不完善引起的问题，我们采用了谨慎修改源码，调整接口调用的方式来进行应对。

综合测试

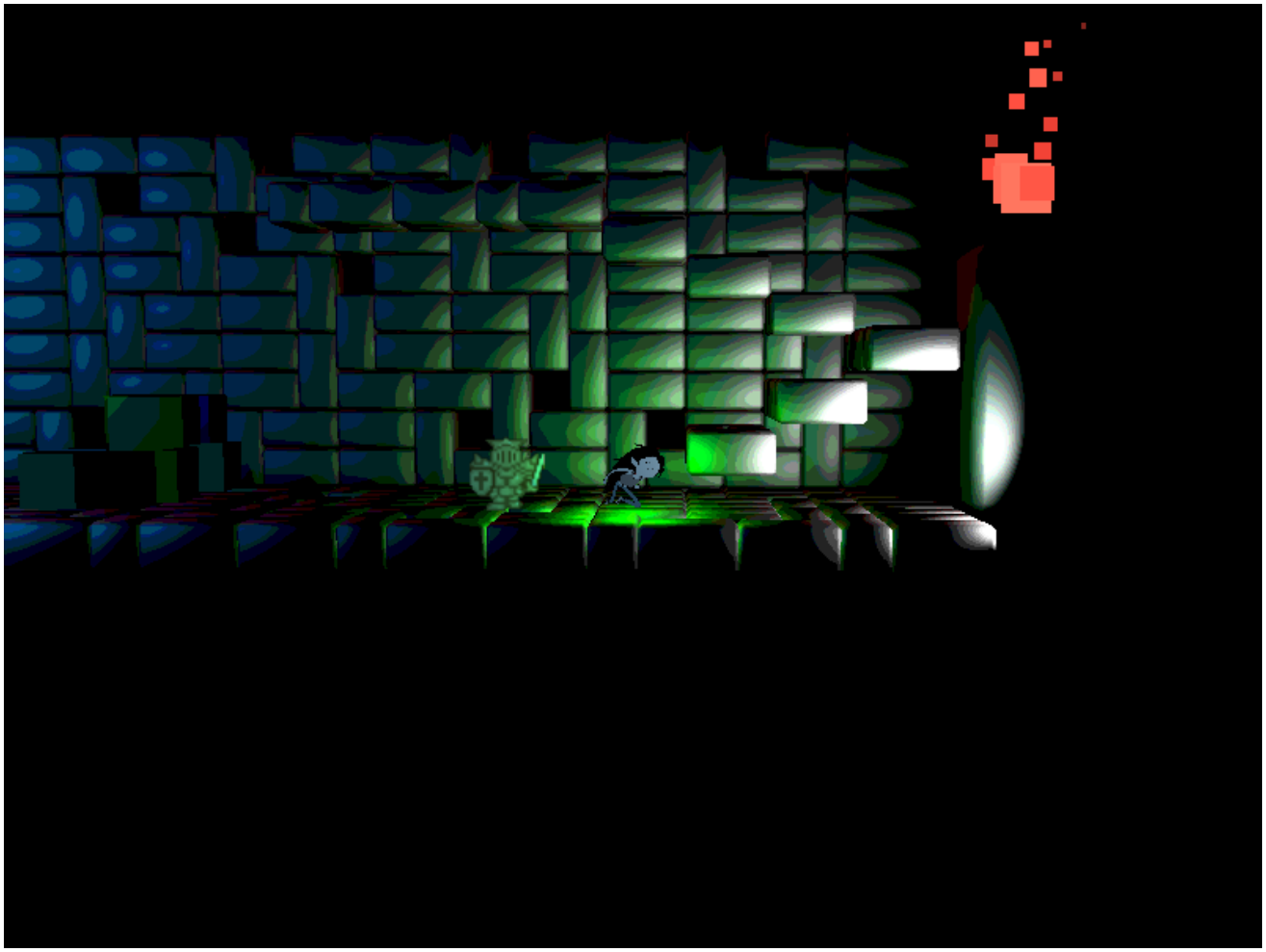
游戏开场场景测试：



玩家键盘控制测试：



NPC自动机响应伴随测试：



交战功能测试：