第二章　　开发环境和相关技术介绍

2.1 开发环境综述

本游戏的开发涉及到了多种工具。开发平台为 microsoft windows 7 32 位，开发 IDE 为 microsoft visual studio 2013，主要的编程语言是 C++，游戏的绘图，输入输出和资源控制等由游戏引擎 HGE 负责，游戏过程中涉及到的物理计算由物理引擎 BOX2D 负责，游戏的地图制作设计由图块编辑器 Tiled 完成。

2.3 游戏引擎 HGE

HGE 的底层使用 DirectX 进行渲染，本小节将先对 DirectX 进行简介。

2.3.1 DirectX 简介

DirectX，（Direct eXtension，简称 DX）[8]是由微软公司创建的多媒体编程接口。由 C++编程语言实现，遵循 COM。被广泛使用于 Microsoft Windows、Microsoft XBOX、Microsoft XBOX 360 和 Microsoft XBOX ONE 电子游戏开发，并且只能支持这些平台。最新版本为 DirectX 12，创建在最新的 Windows 10 上。

DirectX 旨在使基于 Windows 的计算机成为运行和显示具有丰富多媒体元素（例如全色图形、视频、3D 动画[9]和丰富音频）的应用程序的理想平台。DirectX 包括安全和性能更新程序，以及许多涵盖所有技术的新功能。应用程序可以通过使用 DirectX API 来访问这些新功能。

DirectX 被广泛用于 Microsoft Windows、Microsoft Xbox 电子游戏开发，并且只能支持这些平台。除了游戏开发之外，DirectX 亦被用于开发许多虚拟三维图形相关软件。Direct3D 是 DirectX 中最广为应用的子模块，所以有时候这两个名词可以互相代称。

DirectX 主要基于 C++编程语言实现，遵循 COM 架构。

DirectX 是由很多 API 组成的，按照性质分类，可以分为四大部分，显示部分、声音部分、输入部分和网络部分。

2.3.2 游戏引擎 HGE

HGE 是一个硬件加速的 2D 游戏引擎，它是一个富有特性的中间件，可以用于开发任何类型的 2D 游戏。HGE 封装性良好，以至于你仅仅需要关心游戏逻辑，而不需要在意 DirectX，Windows 消息循环等。HGE 架构在 DirectX 8。0 之上，能够跑在大多数的 Windows 系统上。

HGE 的特点主要体现在 5 个方面：

(1) 专业化：专注于 2D 领域

(2) 简单化： 非常容易使用

(3) 技术优势：基于 Direct3D API 有较好的性能和特性

(4) 免费：对于个人或者商业用户都免费，遵循 zlib/libpng license

(5) 代码高度的一致性：代码是否具有一致性，是衡量代码质量的标准之一

HGE 有 3 个抽象层（layers of abstraction）：

(1) 核心函数（Core Functions）：处于核心的函数和例程（routines），被整个系统所依赖。

(2) 辅助类（Helper Classes）：游戏对象相关的类，架构于 HGE 核心函数层之上，辅助用户进行游戏开发。

(3) 创作工具（Authoring Tools）：用于游戏开发的一组工具。

HGE 体系结构如图 2.1 所示。

2.4 物理引擎 BOX2D

Box2D 是一款免费的开源二维物理引擎[10]，由 Erin Catto 使用 C++编写，在 zlib 授权下发布。它已被用于蜡笔物理学、愤怒的小鸟、地狱边境、Rolando、Fantastic Contraption、Incredibots、Tiny Wings 等游戏的开发。

Box2D 是一个用于模拟 2D 刚体物体的 C++引擎。zlib 许可是一个自由软件授权协议，但并非 copyleft。

Box2D 以库的形式提供其对物理世界的支持，它轻量、健壮、高效以及非常的小巧，并且在很多平台上都开辟了很多应用，而且还是开源和免费的。

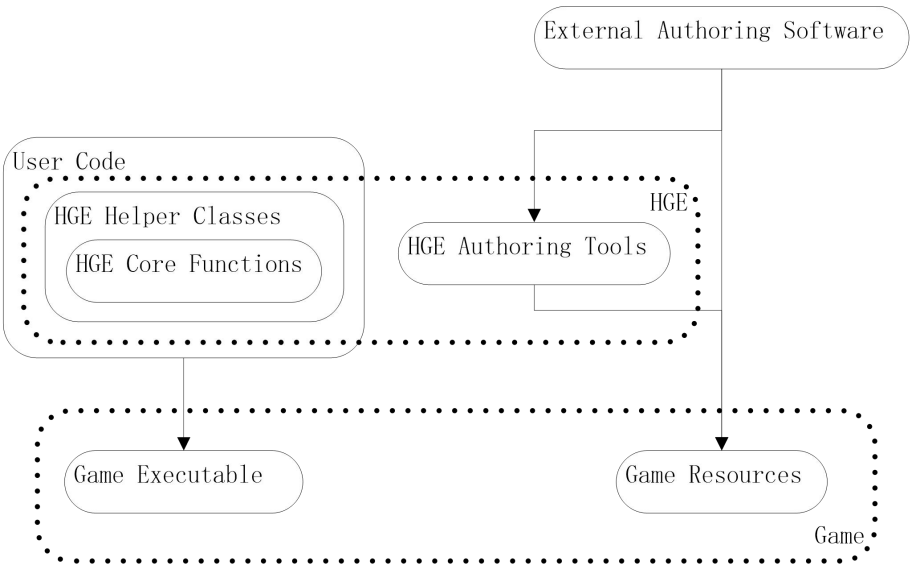

图 2.1 HGE 体系结构

Box2D 核心概念包括以下几点：

(1) 刚体(rigid body)：

一块十分坚硬的物质，它上面的任何两点之间的距离都是完全不变的。一般用物体(body)简单的来描述刚体。

(2) 形状(shape)：

一块严格依附于物体(body)的 2D 碰撞几何结构(collision geometry)。形状具有摩擦(friction)和恢复(restitution)的材料性质。

(3) 约束(constraint)：

一个约束(constraint)就是消除物体自由度的物理连接。在 2D 中，一个物体有 3 个自由度。如果我们把一个物体钉在墙上(像摆锤那样)，那我们就把它约束到了墙上。这样，此物体就只能绕着这个钉子旋转，所以这个约束消除了它 2 个自由度。

(4) 接触约束(contact constraint)：

一个防止刚体穿透，以及用于模拟摩擦(friction)和恢复(restitution)的特殊约束。你永远都不必创建一个接触约束，它们会自动被 Box2D 创建。

(5) 关节(joint)：

它是一种用于把两个或多个物体固定到一起的约束。Box2D 支持的关节类型有：旋转，棱柱，距离等等。关节可以支持限制(limits)和马达(motors)。

(6) 关节限制(joint limit)：

一个关节限制(joint limit)限定了一个关节的运动范围。例如人类的胳膊肘只能做某一范围角度的运动。

(7) 关节马达(joint motor)：

一个关节马达能依照关节的自由度来驱动所连接的物体。例如，你可以使用一个马达来驱动一个肘的旋转。

(8) 世界(world)：

一个物理世界就是物体，形状和约束相互作用的集合。Box2D 支持创建多个世界，但这通常是不必要的。

(9) 求解器(solver)：

物理世界中有一个求解器，用于推进时间和解决接触约束和关节约束。

(10) 连续碰撞(continuous collision)：

使用不连续的时间步来推进刚体的求解器。

2.5 图块编辑器 Tiled

2.5.1 地图编辑器

地图编辑器是一种所见即所得的游戏地图制作工具，它辅助设计和输出地图数据，包括创建、编辑、存储和管理游戏地图数据。

地图编辑器读取和使用游戏资源，并按照游戏程序规约输出相应格式的地图

数据，游戏程序(客户端和服务器)通过地图数据构建游戏场景，将其呈现给用户。地图编辑器的主要功能包括地图制作和地图资源管理两部分。地图制作主要包括地地表生成、地图物体摆放、地图属性设置和地图数据输出；地图资源管理包括地图物体编辑、地图物体属性设置和资源数据输出。这里的地图物体是指用于表示地表、树木、房屋、精灵等摆放在游戏地图上的图片和动画资源。在地图编辑器中，开发人员可以方便地摆放地图物体、构建和修改地图场景、自动判断遮挡关系以及设置地图事件等。地图编辑器通过其直观和简易的操作来简化地图的制作过程，地图编辑器的资源管理功能使得地图资源可以在多个地图中复用，极大地减少地图制作和修改的工作量。因此，很多游戏开发商在项目初期开发出项目的地图编辑器，用于提高游戏的开发效率，减少地图搭建阶段所花费的时间，以缩短项目周期。

2.5.2 图块编辑器 Tiled 简介

Tiled 是基于 Java/Qt 的开源区块地图编辑器，Tiled 支持 2D 和 2.5D 地图及多种区块类型，它将地图保存成一个 xml 文件，借助 xml 的特性使地图可通用于各种游戏平台。Tiled 还支持通过插件来读写地图数据，用户可以方便地自定义地图的输出格式。Tiled 给每个区块命名，并通过区块集(tileset)提供了简单的区块管理工具。Tiled 同样支持地图分层，并且可以为每个层次添加各种属性。除此之外，Tiled 还支持自定义对象图层，用户可以在该层上添加各种数据，这对地图的事件触发设置提供了较好的支持。另外，Tiled 将所有图片都以最小单位区块的大小进行切割，以牺牲地图美观来避免深度排序和图片偏移等复杂的计算，Tiled 编辑器界面如图 2.2。

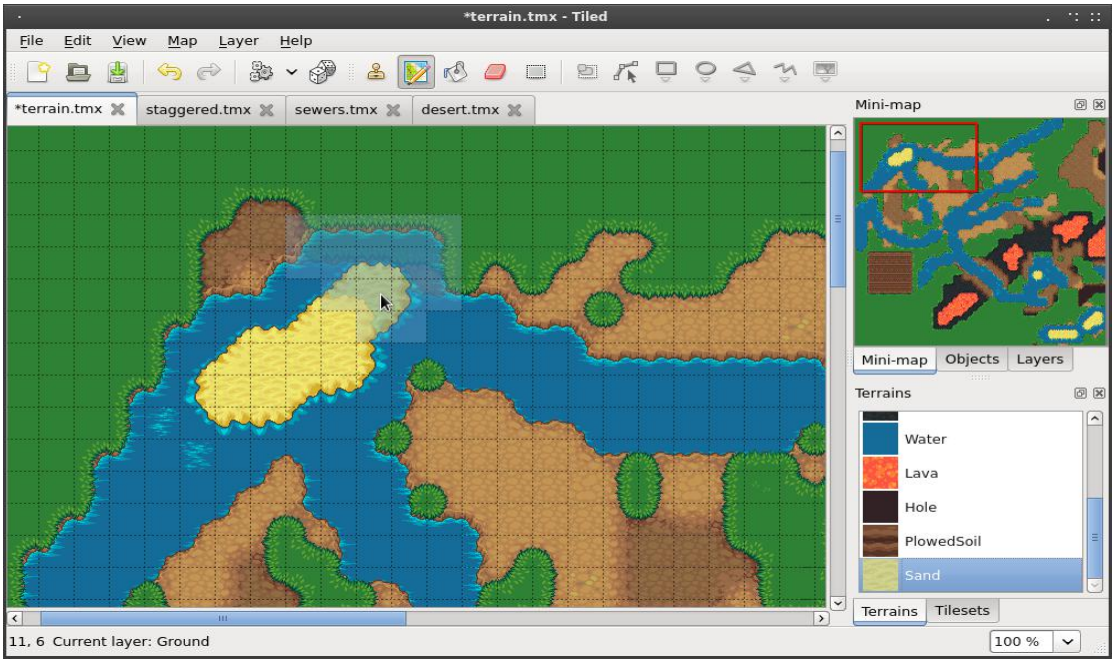

图 2.2　　Tiled 图块编辑器

2.6 本章小结

本章主要对坦克大战游戏软件设计中所用的开发工具和语言做了简要介绍。C++作为使用最为广泛的一种语言，具有简单性、安全性、灵活性等优势；游戏引擎 HGE 为游戏的绘图和输入控制打下基础；物理引擎 Box2D 为游戏提供物理演算；图块编辑器 Tiled 提供了游戏的地图表示，提高开发效率。坦克大战游戏结合使用本章的工具进行开发。

第三章　坦克大战游戏程序设计的理论基础

3.1 坦克大战游戏规则

本次设计的坦克大战的主要游戏规则可以分为以下几点：

(1) 坦克的移动使用上下左右 4 个方向[11]，移动时可能遇到障碍物阻止其继续向当前方向移动。

(2) 坦克攻击时将会按其当前方向在坦克前方产生子弹，子弹可以和大部分游戏元素进行碰撞，对其造成伤害，根据伤害的大小可能会将子弹所碰撞的元素摧毁，子弹碰撞后将自行摧毁。

(3) 玩家进入游戏后可操纵己方坦克移动和攻击。

(4) 游戏中会出现敌方坦克，敌方坦克按特定 AI 进行移动和攻击[12]。

(5) 所有的坦克都有队伍标志，标志相同的互相为盟友，不同的则互相为敌。盟友不会被盟友的子弹所伤害。

(6) 游戏中的部分障碍物可通过连续或非连续的子弹接触破坏。

(7) 游戏里会随机出现不同类型的奖励，坦克接触后将有不同效果。奖励并不总是作用于所接触的坦克[13]。

(8) 当己方坦克死亡时游戏失败，游戏失败时可以重新开始游戏本关卡；当消灭所有敌方坦克后游戏胜利，如果有下一关卡则可进入下一关。

(9) 玩家可使用某些按键进行游戏的暂停，继续和退出。

3.2 地图和关卡设计

3.2.1 地图

地图是坦克大战游戏时除坦克外游戏所呈现的环境，是一个图像显示信息和物理世界信息的综合体。游戏中的地图采用 2D 俯视图，分上下左右 4 个方向。由于地图是屏幕显示元素，所以其长度单位为像素。

地图将使用图块编辑器 Tiled 进行设计而不使用代码硬编，地图的编辑也是一项略为烦琐的过程，使用编辑器有利于提高项目整体效率，同时可以方便的定义参数，调试地图，相对于代码硬编也提高的程序代码的可读性。

一般地图的构成要素主要包括以下几点：

(1) 地图大小。地图的大小决定了游戏时坦克，子弹和奖励等元素位置的合法区间，地图的大小由不可破坏的障碍物所限定。将游戏元素置于地图范围外是无意义且浪费资源的，所有游戏元素都将显示在地图所限定的范围内。另外，游戏地图大小必须符合游戏窗口中可视域的大小，游戏窗口大小将确定为 800x600 像素，考虑到可视域的高会比窗口的小一些，所以地图的高将在此基础上适当减

小。

(2) 地图形状。由于地图是上下左右4个方向的俯视图，所以地图采用宽和高限定的矩形作为地图形状。矩形形状普遍而简单，在设计制作地图时的复杂度也远远低于圆形和不规则多边形。

(3) 地图分块。当使用图块编辑器编辑地图时，地图将被分为若干小的地图块，每个地图块之间的宽高都相同，定义一个良好的分块大小可以使地图编辑更方便，同时也对元素位置有一个较好的把握。考虑到本游戏的地图大小，可以将分块大小设为32个像素。

(4) 地图图像图层。在地图图像图层里可以预先定义一些不会改变的静态图像，如静态背景图，游戏LOGO等，在设计过程中预先画上一些地形图片有利于其他游戏元素位置的确定。

(5) 地图对象图层。地图对象图层中包含自定义地图对象，这些对象通常是不可见的，在图块编辑器中通常是透明的，它们用来定义地图的一些抽象元素，如单位出生点，事件触发点等，对象也有大小，可以自由定义，对象的大小可以作为某些游戏元素的大小，在地图中定义好对象后由程序代码读取并进行相应操作。

(6) 属性。层和对象都可以添加自定义属性，属性包括属性名称和以字符串表示的值。属性可以定义对象或层的各种参数，对象属性偏向于和游戏具体元素相对应，如对象的最大生命值，对象类型等，层属性则偏向于和游戏全局参数，游戏的控制逻辑相对应，如敌人总数，同时出现的敌人数量等。将这些属性从游戏代码中分离开来更有利于制作外部参数控制的游戏，由于更改参数后不需要重新编译链接从而提高了程序的调试效率。

(7) 文件表示形式。地图的文件表示形式为后缀名是".tmx"的文件，此文件实际上是xml格式的文件，在程序中可通过相应的解析库分析从而获取地图中的各种参数和定义。同时，使用xml格式也使得游戏地图很小，节省硬盘空间。

3.2.2 关卡

关卡可以看作是地图加入不同参数和定义后的实例[14]，关卡中涉及到的是具体的游戏元素，本游戏的关卡主要包含以下内容：

(1) 信息对象层。该层包含本关卡的参数信息，保存在层属性之中，包括关卡名称和下一关卡的文件名(不包含后缀名)，关卡名称将在游戏过程中显示，当本关卡游戏胜利时将检查下一关卡文件名，如果存在则在玩家确认后加载下一关卡。该层没有自定义对象。

(2) 敌方出生对象层。该层包含敌方出生点信息和敌方出生的参数。敌方出生点位置使用自定义对象进行定位，即本层中对象的位置即为出生点的位置，同时敌方的队伍标志和类型将由出生点对象的属性进行描述，在本层的层属性里将

会定义本层的敌人总数和最大同时存在的敌人数量。

(3) 玩家对象层。玩家对象层定义了简单的玩家基本信息。玩家对象层中将定义一个有名的对象作为玩家的出生点，玩家对象层的层属性中将定义玩家的最大生命值属性。

(4) 障碍物对象层。该层主要定义障碍物的相关信息。该层没有层属性，每一个层中的对象都代表着一个障碍物，对象属性表示障碍物的类型或特性。对象的大小代表着障碍物的大小。

对不同的游戏对象层设置颜色可以很容易的区分不同层的各种对象，提高关卡编辑的准确性，游戏中每一个关卡都对应着一个.tmx 文件。

3.3 物理世界

物理世界即是由物理引擎所创建的世界，这个世界和通常游戏的绘图世界(即屏幕空间)是独立的，这个世界里的对象称为刚体，刚体拥有自己的物理属性，如质量，摩擦力等等，刚体在物理世界中将遵循一定的物理法则进行运动[15]。

3.3.1 物理坐标系统

因为物理世界是一个相对独立的世界，所以其拥有一个自己的坐标系统，本游戏使用的 Box2D 物理引擎中坐标系的长度单位为米，因为游戏的绘图世界和物理引擎的物理世界的长度单位不一样，所以我们需要定义一个常量用来进行两个世界坐标和长度的转换，通常是定义 32 个像素为 1 米。

3.3.2 物理计算

物理引擎中的物理计算包括计算角速度，线速度，碰撞检测等，给定一个时间间隔，物理引擎便会计算出经过这个时间间隔后整个物理世界中所有刚体的物理属性，包括位置，线速度等，这其间可能会发生刚体之间的碰撞，摩擦等物理行为。

更宏观一点来看，我们的坦克大战中包含的物理计算包括以下两点：

(1) 游戏中实体间的碰撞和穿透。

(2) 子弹的飞行。

在 Box2D 中创建一个简单的物理世界并让其运动起来通常包括以下几点：

(1) 创建一个世界。

每个 Box2D 程序都将从一个世界对象(world object)的创建开始。这是一个管理内存，对象和模拟的中心。

要创建一个世界对象，我们首先需要定义一个世界的包围盒。Box2D 使用包围盒来加速碰撞检测。尺寸并不关键，但合适的尺寸有助于性能。这个包围盒过大总比过小好。

b2AABB worldAABB;

worldAABB.lowerBound.Set(-100.0f, -100.0f);

worldAABB.upperBound.Set(100.0f, 100.0f);

接下来我们定义重力矢量。

b2Vec2 gravity(0.0f, -10.0f);

bool doSleep = true; //当动态物体静止时使它休眠,减少性能开销

现在我们创建世界对象。

b2World world(worldAABB, gravity, doSleep);//在栈上创建 world

那么现在我们有了自己的物理世界。

(2) 创建一个地面

第一步，我们创建地面体。要创建它我们需要一个物体定义(body definition)，通过物体定义我们来指定地面体的初始位置。

b2BodyDef groundBodyDef;

groundBodyDef.position.Set(0.0f, -10.0f);

第二步，将物体定义传给世界对象来创建地面体。世界对象并不保存到物体定义的引用。地面体是作为静态物体(static body)创建的，静态物体之间并没有碰撞，它们是固定的。当一个物体具有零质量的时候 Box2D 就会确定它为静态物体，物体的默认质量是零，所以它们默认就是静态的。

b2Body* ground = world.CreateBody(&groundBodyDef);

第三步，我们创建一个地面的多边形定义。我们使用 SetAsBox 简捷地把地面多边形规定为一个盒子(矩形)形状，盒子的中点就位于父物体的原点上。

b2PolygonDef groundShapeDef;

groundShapeDef.SetAsBox(50.0f, 10.0f);

其中，SetAsBox 函数接收了半个宽度和半个高度，这样的话，地面盒就是 100 个单位宽(x 轴)以及 20 个单位高(y 轴)。

在第四步中，我们在地面体上创建地面多边形，以完成地面体。

groundBody->CreateShape(&groundShapeDef);//创建形状用于碰撞检测等

(3) 创建一个动态物体

首先我们用 CreateBody 创建物体。

b2BodyDef bodyDef;

bodyDef.position.Set(0.0f, 4.0f);

b2Body* body = world.CreateBody(&bodyDef);

接下来我们创建并添加一个多边形形状到物体上。注意我们把密度设置为 1，默认的密度是 0。并且，形状的摩擦设置到了 0。3。形状添加好以后，我们就使用 SetMassFromShapes 方法来命令物体通过形状去计算其自身的质量。这暗示了你可以给单个物体添加一个以上的形状。如果质量计算结果为 0，那么物体

会变成真正的静态。

```
b2PolygonDef shapeDef;
shapeDef.SetAsBox(1.0f, 1.0f);
shapeDef.density = 1.0f;
shapeDef.friction = 0.3f;
body->CreateShape(&shapeDef);
body->SetMassFromShapes();
```

(4) 模拟 Box2D 的世界

我们已经初始化好了地面盒和一个动态盒。现在我们只有少数几个问题需要考虑。Box2D 中有一些数学代码构成的积分器(integrator)，积分器在离散的时间点上模拟物理方程，它将与游戏动画循环一同运行。所以我们需要为 Box2D 选取一个时间步，通常来说游戏物理引擎需要至少 60Hz 的速度，也就是 1/60 的时间步。你可以使用更大的时间步，但是你必须更加小心地为你的世界调整定义。我们也不喜欢时间步变化得太大，所以不要把时间步关联到帧频(除非你真的必须这样做)。直截了当地，这个就是时间步：float32 timeStep = 1.0f / 60.0f；

除了积分器之外，Box2D 中还有约束求解器(constraint solver)。约束求解器用于解决模拟中的所有约束，一次一个。单个的约束会被完美的求解，然而当我们求解一个约束的时候，我们就会稍微耽误另一个。要得到良好的解，我们需要迭代所有约束多次。建议的 Box2D 迭代次数是 10 次。你可以按自己的喜好去调整这个数，但要记得它是速度与质量之间的平衡。更少的迭代会增加性能并降低精度，同样地，更多的迭代会减少性能但提高模拟质量。这是我们选择的迭代次数：

```
int32 iterations = 10;//一个时间步遍历 10 次约束
```

现在我们可以开始模拟循环了，在游戏中模拟循环应该并入游戏循环。每次循环你都应该调用 b2World::Step，通常调用一次就够了，这取决于帧频以及物理时间步。

通过以上几步，一个简单而有活力的物理世界就创建完毕了。

3.4 游戏实体元素

游戏实体元素是游戏的核心元素，也是游戏的基本组成单位，它定义了其在游戏中所需的属性和方法，从基本实体往下可以延伸出多种多样的游戏实体，从而搭建起整个游戏世界[16][17]。

3.4.1 实体属性和方法

实体本身的属性可以分为以下 4 种：

(1) 实体基本属性。实体的基本属性用于实体在程序中的一般逻辑交互，为

实体提供类型辨识，实体的基本属性只包括用字符串表示的实体类型，即该实体或由该实体延伸出的实体的类型名称，这一属性在碰撞检测的实体辨别中十分有用，将是碰撞处理与否的关键。

(2) 实体绘图属性。实体的绘图属性即为实体在绘图世界中的所需的属性，是在屏幕上显示实体所不可缺少的参数，本游戏的绘图功能由HGE游戏引擎所驱动。绘图属性包括实体绘图载体，实体深度，以宽高限定的大小，以二维坐标确定的位置及实体的旋转角度。绘图载体是输入给图形引擎以在屏幕上输出图像而用的；由于这是一个2D游戏，某些情况下允许实体重叠，实体之间的先后绘图顺序便由实体深度所决定，实体越深则绘图顺序越靠后；宽高大小决定了实体最终绘图的范围，这在游戏过程中不会发生改变；坐标则决定最终绘图的位置；由于游戏中只有上下左右4个方向，所以实体的旋转角度只有0，90，180，270这4个度数取值。

(3) 实体物理属性。物理属性为实体和物理引擎交互所需的属性，即代表了实体在物理世界中的刚体的一面。物理属性也就是刚体载体。在物理世界进行物理计算时需要用到这个载体。值得注意的是刚体本身也有大小和表示其位置的坐标，刚体的大小和实体绘图属性中保持一致，坐标则是其在物理世界中的坐标，游戏中需要将刚体坐标和绘图坐标以某种方式统一起来，这将在下一章介绍。

(4) 实体游戏属性。游戏属性即是在游戏环境下实体对外额外表达的属性，用于游戏的交互。这类属性包括实体的当前生命值，最大生命值和是否无敌标志。当一个实体为无敌状态时，它被杀伤类实体(如子弹)碰撞后将不对自身造成伤害。

以上提到具体属性均为基本实体所拥有的属性，由基本实体延伸下来的实体可以拥有自己额外的属性，一般扩展的属性大多是游戏属性。

实体的方法也是建立在实体属性基础上的，除了实体的构造析构外，大部分的方法都是对属性的getter和setter方法，另外还包含以下3个功能性方法：

(1) 设置纹理。即为实体设置其绘图时所用的纹理，这个方法内部实际是调用实体绘图载体的纹理设置方法。

(2) 设置实体线速度。这个方法是用来设置实体的刚体载体在物理世界的线速度的，通常用来实现实体的自动移动(如子弹的飞行)，这个方法的内部实际是调用刚体载体的设置线速度方法。

(3) 撞击方法。此方法用于一定条件下实体发生撞击时使用，传入一个伤害参数，当实体不是无敌状态时该方法按伤害参数减少实体的生命值，按实体生命值是否归零返回不同的返回值。

3.4.2 游戏实体管理

游戏中的实体大多数都是动态生成的，从控制的角度来说一个实体管理模块是必不可少的，这样即利于直接对特定实体或所有实体进行操作，也利于内存的

管理，防止内存泄露[18] [19]。

实体管理的职责是管理维护所有游戏中的实体和创建维护物理世界的运行。

由于这是一个管理模块，在整个游戏过程中仅需要一个实例，所以这个模块以单例的设计模式实现。

除了单例模式所需的一个模块自身实例外，实体管理还包括以下 2 类属性：

(1) 所有实体列表。当产生一个实体时，实体管理便会将其加入到实体列表中，实体销毁时同样会将其移出列表，通过该列表我们还可以检查一个传入的实体是否合法。

(2) 物理世界控制属性。这部分属性包括了创建并运行一个物理世界所需的参数和实现其他物理世界特性的属性：世界载体，物理世界调试绘图，物理世界调试开关控制，碰撞监听和需要删除的刚体列表。世界载体维护物理世界；物理世界调试绘图则用于在绘图世界以某种形式画出物理世界里的那些刚体，这样便于调试物理世界和绘图世界的统一；物理世界调试开关则是决定是否开启调试；碰撞监听收集物理世界的碰撞信息，并可以自定义的回调给外部模块；需删除的刚体列表维护一个已废弃的刚体列表，由于物理引擎的特性导致实体中的刚体载体和绘图载体不能在同一时刻进行销毁，刚体载体的销毁往往略为偏后一个时间点，这里的需删除刚体列表便是用来纪录这些刚体并提供一个删除缓冲的。

实体管理的方法中最核心的是步进方法。该方法进行物理世界的模拟并同步实体在绘图世界的位置等信息，同时还进行废弃刚体的删除等操作，这个方法将在下一章详细描述。

3.4.3 坦克

坦克是由基本实体延伸出的实体，用于表示游戏中的坦克，该种实体包含了本坦克大战中坦克所需的基本属性，并定义了一些易于外部操作坦克的接口。

坦克建立在基本实体之上，还包括以下 3 个属性：

(1) 子弹威力。

(2) 队伍标志。用于区分坦克的阵营，不同标志的坦克互相为敌，同一标志的坦克则属于同盟。

(3) 移动速度。坦克每次移动时移动的像素量。

坦克的方法除了对于其扩展属性的 getter 和 setter 外还包括 2 个对外接口方法：

(1) 开火。使用此方法坦克将立即向其前方开火，其本质上是在其面朝的方向前方动态生成一个飞行的子弹。

(2) 移动。传入一个方向参数和速度参数(可选)，坦克将按参数向指定方向移动指定像素，速度参数不指定时将使用坦克内部的移动速度属性。

由于敌方坦克受到伤害后会根据剩余生命值百分比改变自己的纹理，所以敌

方坦克实体是由坦克实体延伸而来的，它将重载撞击方法以实现此功能。同时敌方坦克可以根据参数变化的不同进一步进行坦克的分类，所以敌方坦克实体在构造时将增加种类这一参数。

3.4.4 子弹

游戏中的子弹是唯一可以对实体造成伤害的元素，子弹由坦克开火产生并沿着某一方向一直飞行，直到与其他实体发生碰撞而毁灭。

子弹由基本实体延伸而来，其在构造时定义为一个可穿透的实体，拥有这一特性的实体其刚体在发生碰撞时会穿透所碰撞的刚体，子弹的这一特性用于子弹穿透奖励。

子弹包括以下 2 个属性：

(1) 所有者。当子弹和其他坦克碰撞时我们需要知道这是不是一个队友伤害，即子弹来自和碰撞者同一个队伍的坦克，这时是无法造成伤害的，每个子弹纪录自己的所有者可以简单高效的完成这一检测。

(2) 攻击力。即为子弹可以对其他实体造成的伤害量，这一属性和坦克的子弹威力属性是一致的，在坦克开火时由坦克指定。

子弹没有特殊的方法，其方法只是对其属性的 getter 和 setter。

3.4.5 障碍物

障碍物是游戏基本实体的简单延伸，用于表示游戏中阻碍坦克运动的障碍物，有的障碍物可通过子弹破坏而有的则是无敌的(比如地图边界)，障碍物没有定义扩展的属性和方法。

3.4.6 奖励

奖励是在游戏中以一定时间间隔随机出现的实体，当坦克与奖励发生碰撞时会产生一定的效果。奖励在一定时间内若没有与坦克碰撞则会自动销毁。

基本奖励实体延伸自游戏基本实体，奖励实体是可穿透的，同时也是无敌的。基本奖励实体定义了与坦克碰撞后行为的接口，所有的具体奖励实体需要实现这个接口以实现各种奖励不同的效果。

游戏中的奖励实体种类包括提高攻击力，回复生命值，在一段时间内提高移动速度，在一段时间内无敌和玩家分数加成。

3.5 游戏状态管理

游戏的主循环是依靠回调来完成游戏过程的，包括桢回调和渲染回调。桢回调用于处理游戏的逻辑事件和计算(包括键盘等的输入处理)，渲染回调用于处理游戏的绘图。由于游戏可能有菜单界面，游戏界面等多个界面，将所有界面的所有逻辑(绘图)代码全部写在游戏的主要桢(渲染)回调中将会大大提高的复杂度并

降低代码可读性，给游戏的维护和调试带来不可避免的麻烦。

为了避免这样的代码灾难，游戏可以将界面抽象为状态[20][21]，每次的回调都只进行某一状态(运行状态)的相关处理，由一个全局的实例管理状态，需要运行的状态可以随时由当前状态改变。

这样所诞生的就是游戏状态管理模块，这是一个单例模块，其中包含着一个状态栈，提供获取当前状态，改变当前状态，推入推出状态和运行当前状态的桢处理或渲染处理流程等接口。

另外还需要的就是游戏状态模块。基本游戏状态模块提供了桢处理， 渲染处理和状态进入等接口，自定义游戏状态需要实现前两个接口，状态进入接口则是可选的，用于初始化一个状态。

使用状态管理并编写自定义游戏状态可以使游戏流程更加的清晰，更有利于游戏状态管理。使用状态管理的代码将非常简洁：

```
GameStateManager* g_gsmanager;
bool FrameFunc() //游戏主要桢回调处理
{
    g_gsmanager->DoThink();
}
bool RenderFunc() //游戏主要渲染回调处理
{
    g_gsmanager->DoRender();
}
```

3.5.1 主菜单

主菜单状态用于展示菜单项和处理玩家选择。使用 GUI 组织标题和菜单项，菜单项包括单人游戏，设置和退出等，玩家选择单人游戏后将会创建一个游戏中状态并告知状态管理推入游戏中状态；玩家选择设置菜单项后将创建一个设置状态，状态管理同样会推入设置状态；若玩家退出，则直接退出游戏。

3.5.2 简单设置菜单

设置菜单状态非常简单，仅可以提供背景音乐大小的调节功能，同样是使用 GUI 组织标题和设置选项，玩家将通过按键来调节背景音乐或者返回主菜单，返回时会推出当前状态。由于主菜单进入时是推入设置状态，这里返回后运行的状态为主菜单状态。

3.5.3 游戏中状态

该状态是游戏的核心状态，控制着游戏内容的载入和运行。该状态的内容非常多，包括载入关卡，初始化玩家、障碍物、敌方、奖励、人工智能，碰撞处理，

游戏胜败判定，游戏信息显示等。游戏中玩家可通过按键随时返回主菜单，这时是推出当前状态，由于主菜单到游戏中状态使用的是推入，此时的运行中状态便是主菜单状态；通过按键暂停或继续游戏，这并不影响游戏状态而是在游戏中状态里处理；在胜利后进入下一关卡时也不改变状态，只是重置游戏元素并重新载入关卡进行初始化。

3.6 游戏输入控制

游戏的输入控制集中在游戏状态的桢处理过程中，主要包括鼠标感应选择菜单项和键盘控制游戏 2 个方面。在任意状态下都可使用特定按键返回上一状态，在游戏中状态里也只定义了几个简单按键用来控制游戏的进行。本游戏的输入控制由游戏引擎 HGE 所驱动。

3.7 游戏资源管理

游戏的资源可以分为纹理，粒子效果，音效等，资源是和游戏主逻辑相互独立的，所以资源在游戏中的时候最好不要以硬编的形式出现。本游戏中使用一个全局的资源管理模块控制游戏的各种资源的载入和使用，游戏的资源通过一个外部的脚本文件定义，该文件定义了资源种类和资源名到资源文件名和资源参数属性的映射，很好的分离了游戏和资源使用。资源管理模块也是以单例模式实现的，模块在初始化的时候便读取脚本文件并缓存资源以期提高游戏速度，在游戏代码的各个角落可以轻松的使用所需要的资源。游戏资源管理依赖于游戏引擎 HGE。

以下是脚本文件中简单的资源定义格式：

```
Texture cursor //定义光标的纹理
{
        filename=cursor.png
}
```

3.8 音效管理

适当的音效可以为一个游戏锦上添花，本游戏使用一个单例实现的音效管理模块进行声音的处理。对于背景音乐，游戏中只允许同时播放一个，如果提出播放请求的时候正在播放其他的背景音乐，那么管理模块将会暂停当前的播放而播放请求的音乐；对于音效则没有同时播放数量的限制。音效管理模块还提供背景音乐的暂停，继续播放功能，音量调节功能和渐变调整，其中渐变调整是指在一定时间间隔内渐变改变背景音乐音量到指定值，可用于音乐的渐入渐出效果。音效管理模块依赖于 HGE 游戏引擎的 BASS 模块。

3.9 粒子系统

粒子系统表示计算机图形学中模拟一些特定的模糊现象的技术[22]，而这些现

象用其它传统的渲染技术难以实现真实感。经常使用粒子系统模拟的现象有火、爆炸、烟、水流、火花、落叶、云、雾、雪、尘、流星尾迹或者像发光轨迹这样的抽象视觉效果等等。简单的说，一个粒子系统就是用来模拟一种特定的效果的。

游戏中的粒子系统同样属于一种资源，由游戏资源管理模块载入。在游戏中，一个单例实现的粒子系统管理模块进行所有粒子系统的控制，该模块提供了生成粒子系统，生成跟随实体的粒子系统，更新显示粒子系统和销毁粒子系统的方法，便于在游戏的各处控制粒子系统的产生和销毁。粒子系统的制作可以使用 HGE 游戏引擎提供的粒子系统编辑器来进行，如图 3.1 所示。

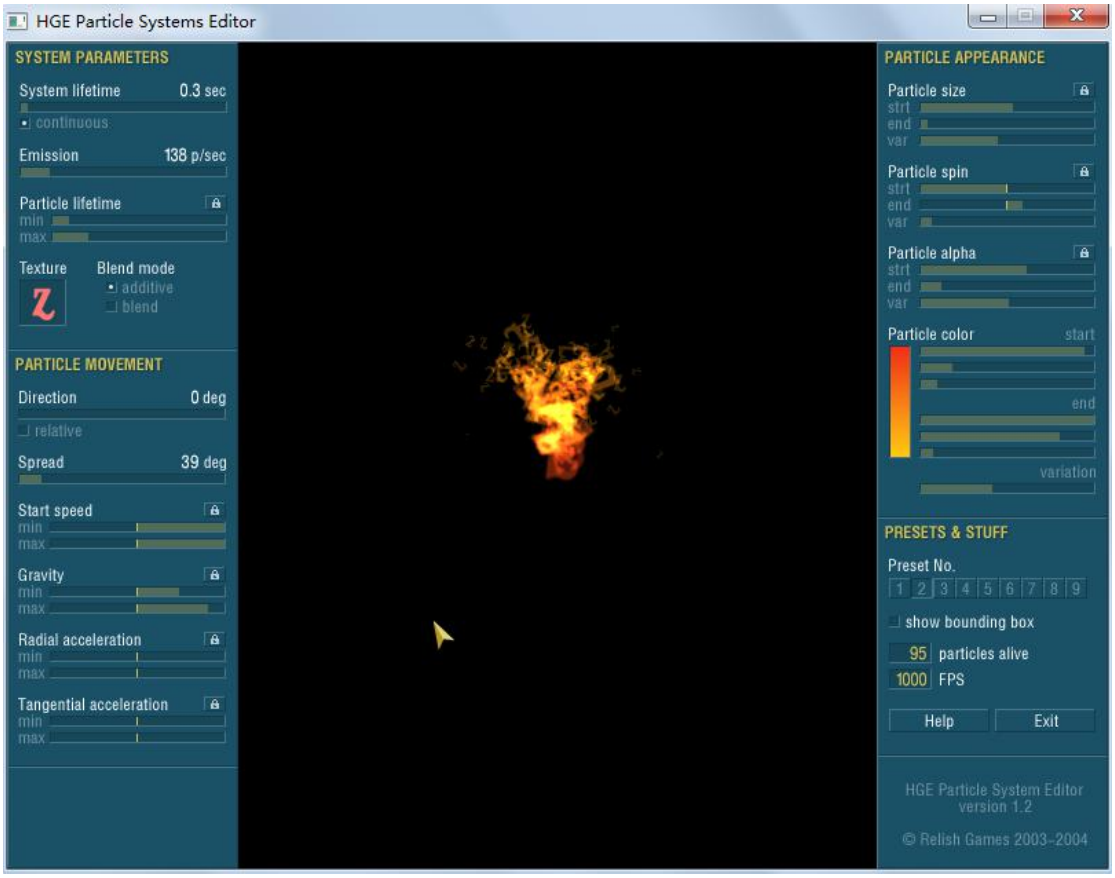

图 3.1　　粒子系统编辑器

3.10 游戏 UI

图形用户界面(GUI)是一种人与计算机通信的界面显示格式[23]，允许用户使用鼠标等输入设备操纵屏幕上的图标或菜单选项，以选择命令、调用文件、启动程序或执行其它一些日常任务。与通过键盘输入文本或字符命令来完成例行任务的字符界面相比，图形用户界面有许多优点。图形用户界面由窗口、下拉菜单、对话框及其相应的控制机制构成，在各种新式应用程序中都是标准化的，即相同的操作总是以同样的方式来完成，在图形用户界面，用户看到和操作的都是图形

对象，应用的是计算机图形学的技术。

游戏 UI 也和一般程序的 UI 没有很大区别，本游戏的 UI 部分集中在主菜单和设置菜单状态，主要体现为可与用户鼠标进行交互的菜单项，当鼠标移动到菜单项上时会有音效和选中的动画效果，当菜单初始化和退出时菜单项会以一定的间隔显示或消失，同时伴有动画效果[24]。

3.11 人工智能

由于需要对不论敌方还是我方的坦克进行控制，游戏中也有着人工智能(AI)模块[25]。在游戏的每一桢中进行 AI 的运行(“思考”)，从而达到用 AI 控制坦克的效果。

基本 AI 模块只有 1 个基本属性：AI 所控制的坦克。AI 将通过坦克实体的对外接口进行坦克的移动和开火控制。

基本 AI 模块还定义了 AI 运行的接口”思考”和 AI 所控制的坦克和其他实体碰撞时的回调，从基本 AI 模块延伸出去的具体 AI 种类需要实现其自己的”思考”接口，从而实现多样化行为的 AI。

游戏中的 AI 种类主要有以下 2 种：

(1) 玩家 AI。考虑到游戏中玩家是操纵坦克的一方，AI 也是用来操纵坦克的，所以玩家的控制机能可以抽象为一种特殊的 AI，在 AI 的”思考”过程加入输入设备的检测和控制流程即实现了玩家 AI。

(2) 随机 AI。随机 AI 的移动行为和开火行为都是随机的，随机 AI 有预定义的移动周期记数器和开火周期记数器。当新的移动周期开始时 AI 会随机选取一个方向并在接下来的周期中持续向这个方向移动；AI 在”思考”时会有一定几率决定是否开火，当决定开火且新的开火周期开始时 AI 才会控制坦克进行开火。

此外，利用人工智能技术还可开发基于状态机的和基于智能体[26] [27]的 AI，这类 AI 设计复杂但是可以达到提高游戏丰富性的效果。这类 AI 往往要进行策略判定，寻路[28] [29] [30] 等行为。

3.12 本章小结

本章介绍了坦克大战游戏程序设计的理论基础，从坦克大战的游戏规则开始逐步简要剖析了实现一个坦克大战游戏所需要的各种元素。而对于坦克大战游戏，在各种元素之中处于核心地位的则是关卡，实体和游戏状态。我们所要做的是将各种游戏元素有机的结合起来，构成最终的坦克大战游戏。在此基础上我们就可以方便的进行坦克游戏的扩展，如增加新的关卡，新的敌方坦克类型，新的奖励，新的障碍物类型和新的 AI 等。

第四章　　坦克大战游戏的设计与实现

前一章中我们已经分析了设计一个坦克大战游戏所需要的理论基础，接下来我们需要对系统进行设计和规划，实现相应的功能。本游戏含有主菜单，设置菜单和游戏中 3 个界面，拥有调节背景音乐音量，操纵坦克进行游戏，胜负判定和关卡推进等功能，本章将会对这些游戏功能的实现和各个游戏功能模块的设计与交互做详细介绍。

4.1 系统功能划分

坦克大战游戏程序是基于游戏状态进行游戏的，游戏中包含了若干游戏状态，每一个状态显示不同的界面，接受不同的输入，提供不同的功能，使用状态机制不仅可以提高游戏编写的效率，还提高了代码可读性，为以后的维护提供了良好的条件。本游戏的主体功能划分可用图 4.1 来表示。

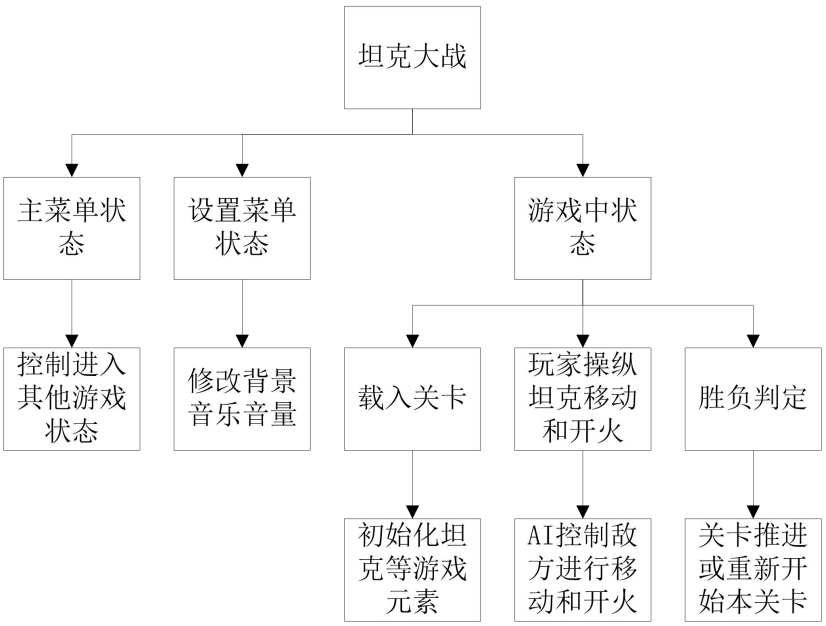

图 4.1　　坦克大战系统功能

4.2 类 GamePlay

该类是游戏核心类之一，是一个较为基础的综合类，封装了 HGE 游戏引擎的绘图控制和输入控制部分，并且提供了对图块编辑器 Tiled 生成的关卡文件的载入和使用功能。整个游戏的初始化由此开始。该类以单例的设计模式实现。其简要结构如图 4.2 所示。

4.2.1 HGE 游戏引擎绘图控制部分

这一部分封装的数据成员和成员函数用于调用 HGE 游戏引擎绘图和控制接口，同时也对 HGE 游戏引擎的接口做了简化处理，使用更加方便。该部分的结构如表 4.1 所示。

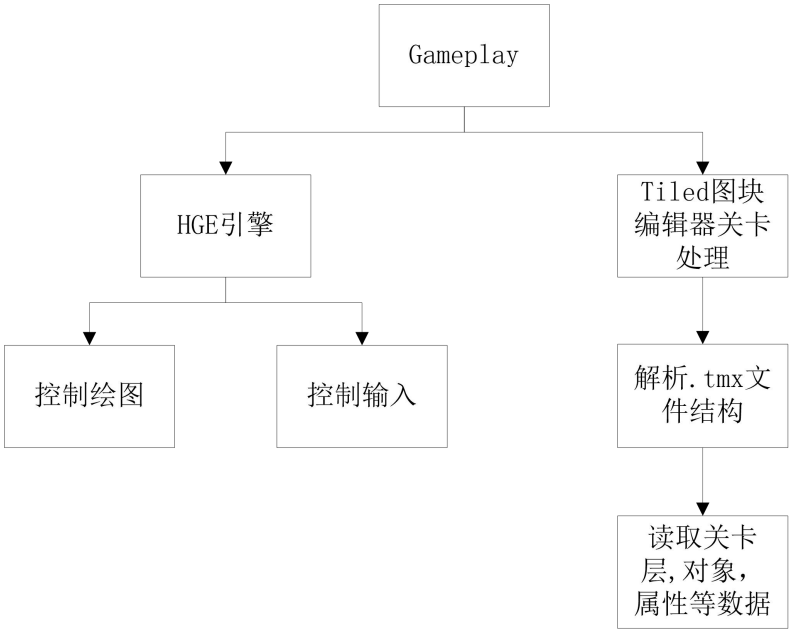

图 4.2　　类 GamePlay 的结构

接下来将对表中的某些重要结构做详细描述。

(1) init()初始化函数

该函数对 HGE 游戏引擎进行初始化，从而建立起整个游戏的绘图机制和输入响应处理机制。其流程图如图 4.3 所示。

(2) 预定义字体对象

使用 GamePlay 内置的预定义字体对象渲染字符串，用户就无需自行定义 HGE 游戏引擎的字体对象进行渲染，这在需要渲染全局字符串信息时方便有效，同时也在一定程度上隔离用户和 HGE 游戏引擎。

用户提供预定义字符串信息将保存在结构体 FontRenderInfo 内，该结构体包括字符串值，字符串位置的二维坐标和字符串渲染时的对齐方式这 3 个信息，对齐方式包括左对齐，右对齐和居中对齐。

使用预定义字体对象进行自定义字符串渲染时的流程如图 4.4 所示。

4.2.2 Tiled 图块编辑器关卡的解析部分

这一部分封装的数据成员和成员函数用于分析 Tiled 图块编辑器生成的 tmx 文件，提供了获取层，对象，属性的功能，并提供一些便捷接口供外部使用。该部分的结构如表 4.2 所示。

表 4.1　　HGE 游戏引擎绘图控制部分结构

类别	声明	作用
数据成员	HGE* m_hge;	是游戏引擎 HGE 其本身的工作接口，对 HGE 的一切操作需要通过这个接口来进行
	hgeFont* m_text;	定义一个 HGE 字体对象，用于 GamePlay 为外部提供预定义的字符串渲染接口功能
	std::vector<FontRenderInfo> m_strtorender;	定义一个向量集合，用于保存在下一渲染调用时需要渲染的字符串信息
	bool Init(hgeCallback framefunc, hgeCallback renderfunc);	初始化函数，用于初始化 HGE 游戏引擎和 GamePlay 预定义的字体对象
	void Run();	进入游戏主循环，游戏内的一切都发生在这里
	void RenderCustomString();	提供自定义字符串的绘制，使用内部字体对象进行绘图
	void PrepareRenderString(const char* str, float x, float y, int align = HGETEXT_LEFT);	添加用户自定义渲染字符串信息的接口，用户自定义字符串只会在下一渲染调用时渲染一次
	bool GetKeyState(int key);	获取某一特定键的状态，若键处于激活时返回 true

当成功载入 tmx 文件后，使用其他的便捷接口可以获取所有的关卡信息，以便进一步进行处理。

4.3 类 ResManager

该类进行所有游戏资源的管理，是一个单例模式实现的类，其内部封装了一个用于资源管理的 HGE 辅助类，资源的载入和使用均通过这个辅助类进行。该类的构成比较简单，其结构如表 4.3 所示。

游戏资源的定义通过一个资源脚本实现， ResManager 在其初始化时先会解析资源脚本，然后预载入所有资源以便让后续游戏流畅运行。

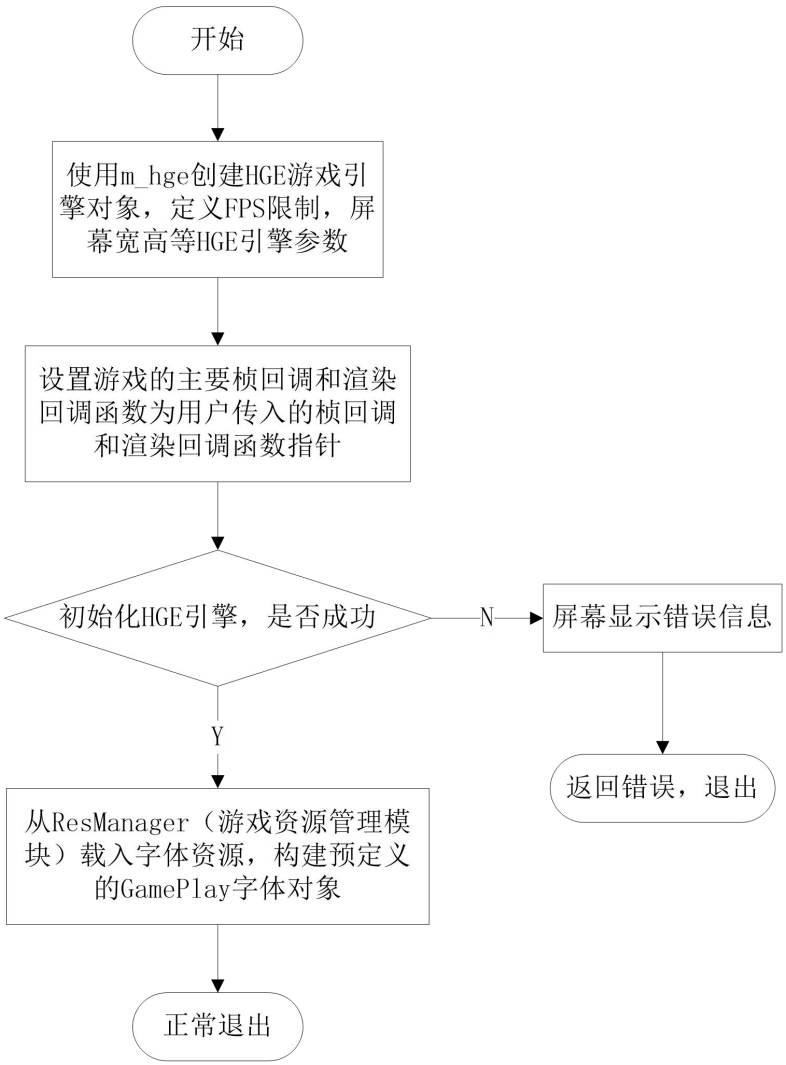

图 4.3　　GamePlay 的 init 初始化函数流程

资源脚本的格式如下：

```
ResourceType ResourceName : BaseResourceName
{
        Parameter1=Value1
        Parameter2=Value2
        ParameterN=ValueN
}
```

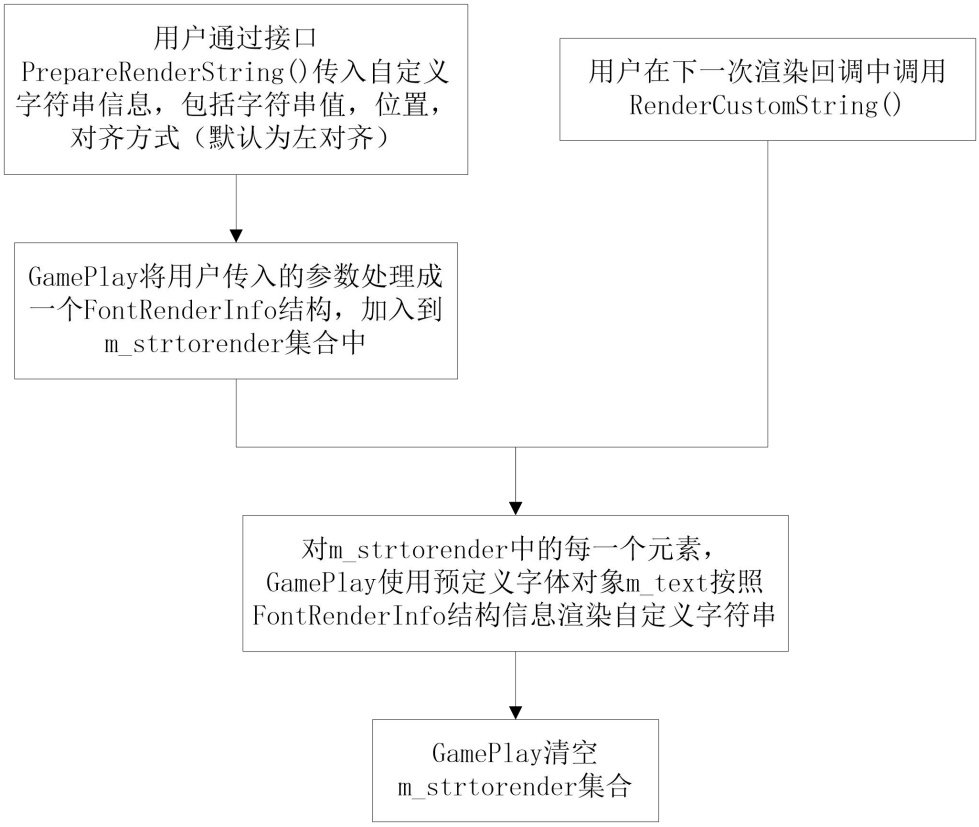

图 4.4　　GamePlay 预定义字体渲染流程

以下是一个音效资源的定义：

```
Sound menu_touch
{
    filename=menu_touch.wav
}
```

对于不同类型的资源其参数通常也不同，资源之间可以互相引用，某一项资源可能需要其他很多资源共同组成，以下是一个粒子系统的资源定义，总共需要 3 个资源组成：

```
Texture particles //粒子效果纹理
{
    filename=particles.png
}
Sprite ps_tank_spawn //粒子系统绘图载体 sprite
{
    texture=particles
    rect=32,64,32,32
    hotspot=16,16
```

```
    blendmode=COLORMUL,ALPHAADD,NOZWRITE
}
Particle tank_spawn //粒子系统
{
    filename=tank_spawn.psi
    sprite=ps_tank_spawn
}
```

表 4.2　Tiled 图块编辑器关卡解析部分结构

类型	声明	作用
数据成员	hgeTileMapXML* m_tilemap;	用来解析 tmx 文件的对象，内部保存解析的结果(如有哪些层，层属性，层对象，对象属性等)，供其他 getter 和 setter 方法使用
成员函数	bool LoadTileMap(const char* filename, const char* mapname);	载入 tmx 文件的接口，内部使用 m_tilemap 进行文件解析
	tilemapxml::ObjectPtr GetObjectInTile(const char* mapname, char* layername, char* objname);	获取特定图的特定层中的特定对象，若层存在且对象存在则返回有效的对象指针
	bool GetObjectInLayer(const char* mapname, char* layername, std::vector<ObjectPtr>& out);	获取特定图的特定层中的所有对象，对象将保存在一个向量集合中被返回
	MapPtr GetMapPTR(const char* mapname);	获取特定图，外部可使用图的相关接口对图进行操作。
	char* GetLayerProperty(const char* mapname, char* layername, char* proname);	获取特定图的特定层的特定属性的值，以字符串形式返回

表 4.3　　ResManager 类的结构

类型	声明	作用
数据成员	hgeResourceManager* m_resmanager;	HGE 资源管理辅助类对象，所有的资源管理实际上由它进行
成员函数	HTEXTURE GetTexture(const char* name);	通过特定资源标识名获取纹理资源
	hgeSprite* GetSprite(const char* name);	通过特定资源标识名获取 Sprite 资源
	hgeFont* GetFont(const char* name);	通过特定资源标识名获取字体资源
	HEFFECT GetEffect(const char* name);	通过特定资源标识名获取音效资源
	hgeParticleSystem* GetParticleSystem(const char* name);	通过特定资源标识名获取粒子系统资源

4.4 类 AudioManager

该类是控制音乐音效播放的管理模块，使用单例的设计模式实现，主要用于播放音效和控制背景音乐的播放。该类实际上是封装了 HGE 游戏引擎的音效功能部分。类的结构如表 4.4 所示。

在 HGE 游戏引擎的音效功能中，如果要对一个已经播放的音乐或音效进行操作则必须使用其所在的频道进行操作，由于我们会对游戏的背景音乐进行暂停、继续、停止等操作，所以我们需要在该类里保存背景音乐成功播放后返回的频道对象。

游戏中的音效可能随时产生，因此音效不会限制同时播放的数量，但一般游戏中的背景音乐在同一时刻只能有一个，所以在调用 PlayBGM()时如果当前正在播放背景音乐，新的背景音乐将会替换掉旧的音乐。

4.5 类 ParticleManager

该类是控制粒子系统产生和更新的管理类，由单例模式实现，内部实际上封装了 HGE 游戏引擎的粒子系统相关功能，除此以外该类还针对游戏实体加入了扩展功能，方便实现实体方面的特殊效果。类 ParticleManager 的主要结构如表 4.5 所示。

表 4.4　　AudioManager 类的结构

类型	声明	作用
数据成员	HGE* m_hge;	一个已初始化的 HGE 游戏引擎接口，由 GamePlay 提供，用于使用 HGE 音效部分功能
	HEFFECT m_effbgm;	游戏背景音乐的音效对象,用于播放背景音乐时新旧音乐的比较
	HCHANNEL m_chbgm;	游戏背景音乐的频道对象,在背景音乐成功播放后会得到这个对象，用于控制频道
	int m_bgmvolumn;	保存背景音乐的音量
成员函数	int GetBGMVolumn();	获取当前背景音乐音量
	void SetBGMVolumn(int v);	更改当前背景音乐音量
	void PlayEffect(HEFFECT eff);	播放音效
	void PlayBGM(HEFFECT bgm);	播放背景音乐
	void PauseBGM();	暂停背景音乐
	void ResumeBGM();	继续播放背景音乐
	void StopBGM();	停止播放背景音乐
	void BGMSlide(float t, int v);	对背景音乐进行渐变处理,在一段时间内使得背景音乐的音量从当前渐变到某一值

通过该类可以生成普通的粒子系统和跟随实体的粒子系统,后者是为了满足某些情况下需要粒子系统移动的需求而产生的。一般而言，用户只需要使用 SpawnParticle()接口来创建粒子系统然后在其程序代码的桢回调和渲染回调中调用 UpdateParticle()和 RenderParticle()便可轻松创建粒子效果，粒子系统一般都有自己的生命周期，无需手动释放。

跟随实体移动的粒子系统的运作使用结构体 MovePartData 纪录其所需要的信息。该结构体包括了粒子系统本身，需要跟随的实体和是否使用”尾随”效果这 3 个信息，其中”尾随”效果是指当实体移动到新位置时原先位置粒子效果不立刻消失，这样当实体快速移动后粒子效果就如同拖尾跟随一般。不使用尾随效果的粒子系统将会覆盖在实体上随实体移动。

表4.5　ParticleManager类的主要结构

类型	声明	作用
数据成员	hgeParticleManager* m_particlemanager;	HGE游戏引擎粒子系统管理对象，粒子系统的大量操作的实际执行者
	std::list<MovePartData> m_movepartlist;	用于纪录需要跟随实体移动的粒子系统的信息
成员函数	bool SpawnParticle(char* partname, float x, float y);	在指定位置生成指定名称的粒子效果
	bool SpawnParticleFollowEntity(char* partname, Entity* ent, bool tail = true);	在指定实体上生成指定的粒子效果，粒子效果将跟随实体一起移动
	void UpdateParticle();	更新所有粒子系统，在游戏桢回调中调用
	void RenderParticle();	渲染所有粒子系统，在游戏渲染回调中调用
	void KillParticle(hgeParticleSystem* ps);	销毁一个粒子系统
	void KillParticleFromFollowedEnt(Entity* ent);	销毁一个跟随实体移动的粒子系统
	void KillAllParticle();	销毁所有粒子系统

跟随实体移动的粒子系统的运作流程图如图4.5所示。

4.6 游戏实体类

游戏实体类包括Entity类，Entity类的延伸子类和Entity的管理类，它们是游戏的核心类，Entity类及其子类是游戏元素的最直接表现，坦克大战游戏内的互动大多数依靠游戏实体类来完成。

4.6.1 类Entity

这是所有实体类的基类，是最基础的游戏元素的定义，由于游戏中使用物理引擎进行物理计算，所以实体必须同时包含绘图世界载体和物理世界载体的信息并将其统一起来，所以该类的设计会比较复杂。该类的设计思路遵循第三章中”

游戏实体元素”这一节所阐述的内容，在此不在赘述。Entity 类的主要结构如表 4.6 所示。

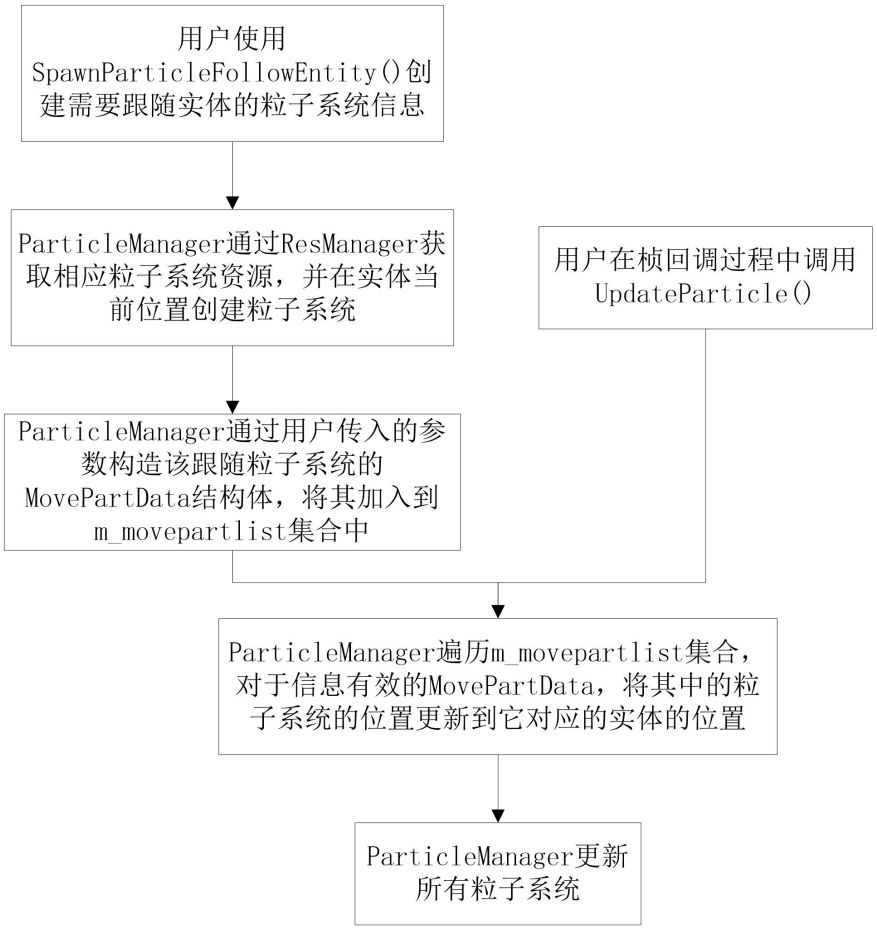

图 4.5　　跟随实体的粒子系统的运作流程

接下来将对表中的重要结构和过程进行详细阐述。

(1) 实体的构造

由表 4.6 可知，实体的构造函数有许多的可选参数，通过不同参数的组合可以实现多种类型和特性的实体，但有些参数在实际中并不常用，对于这些参数构造函数都提供了一个默认值，用于适应大部分默认的情况而不用用户自己操心。其中一些主要参数的含义如表 4.7 所示。

由于实体的创建大都是从读取关卡文件（后缀名为.tmx 的文件）的数据中进行，在 Tiled 图块编辑器中一个对象的位置被表示为其左上角的 x，y 坐标位置，所以 Entity 类将构造函数的 x，y 设定为左上角坐标以方便配合 Tiled 图块编辑器进行外部调用，但是在 Entity 类的内部这些坐标将会被转换为表示实体中心位置的中心 x，y 坐标以便于 HGE 图形引擎进行实体的渲染。当实际使用 Entity 类时不必考虑坐标转换。

表 4.6　Entity 类的主要结构

类型	声明	作用
数据成员	std::string m_type;	实体基本属性，表示实体的类型，将各个实体子类区分开，主要用于碰撞检测类型识别
	hgeSprite* m_spr;	实体在绘图世界的载体
	b2Body* m_b2body;	实体在物理世界的载体
	int m_hp;	实体的当前生命值
	int m_maxhp;	实体的最大生命值
	bool m_isinvincible;	实体是否处于无敌状态
成员函数	Entity(float x, float y, float width, float height, float rot = 0, bool b2donly = false, bool isstatic = true, bool through = false, std::string& type = std::string("entity"));	实体的构造函数，拥有众多参数控制实体的产生，大多数的参数包含默认值。实体的初始化也在这里进行。
	void Destory();	销毁实体本身，调用将会立即销毁实体在绘图世界的信息，实体在物理世界的信息由 EntityManager 销毁
	virtual bool Hit(int attack);	使实体受到一定的伤害，更新实体生命值

实体的构造过程如图 4.6 所示。

构造过程可以简单分为两个部分：一是根据传入的参数构建实体自身的绘图载体和物理载体，其中绘图载体可以选择不构建，这样可以生成一个“隐形”的实体，在实现不可见地图边界、空气墙等效果时很有用；二是向 EntityManager（全局实体管理类）注册自己，方便其进行统一管理。

(2) 实体的销毁

由于物理引擎的特性，实体的物理载体和绘图载体不能在同一时刻销毁，因此需要第三方(即 EntityManager，全局实体管理模块)来控制这个流程，这就增加了实体销毁的复杂度，实体销毁的流程如图 4.7 所示。

实体销毁的核心结构是 EntityManager 内部保存的一个待摧毁刚体列表。调用 EntityManager 的销毁接口是整个销毁过程的第一步，然后是实际销毁实体的

绘图部分，维护待摧毁列表，由于 Box2D 引擎的特性（不允许物理模拟的时候销毁任何刚体），销毁实体的物理部分（即刚体）则是在下一次物理模拟之后才进行，销毁刚体的进行方式是按照待摧毁列表逐个销毁。在销毁了刚体之后，整个实体的资源就被完全释放了。

表 4.7　　实体构造函数的主要参数

参数	默认值	意义
bool b2donly	False	实体是否只有物理部分无绘图部分,这可以用来创建一个不可见但有物理状态的实体，比如空气墙，地图边界等
bool isstatic	true	实体是否是静态的,静态实体可以发生碰撞,但是在碰撞中不会移动
bool through	False	实体是否可穿越,可穿越实体可以发生碰撞,但是在碰撞后不停止运动

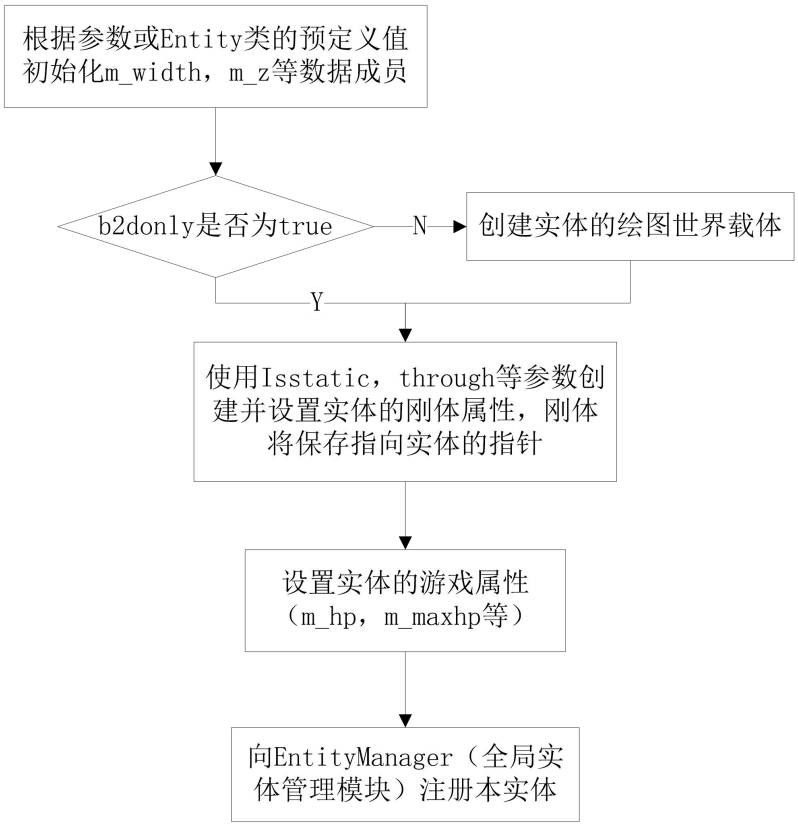

图 4.6　　Entity 的构造过程

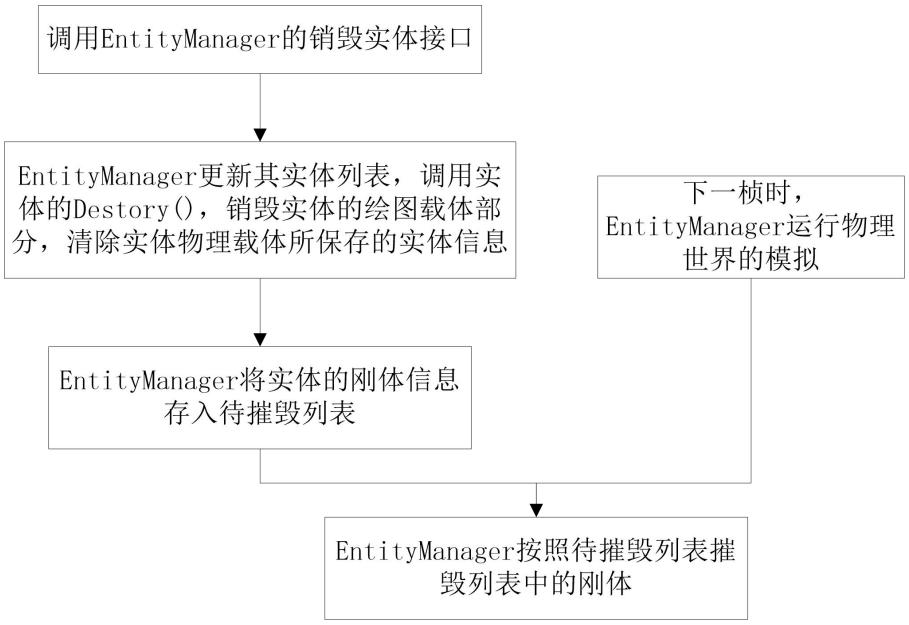

图 4.7　　Entity 的销毁过程

4.6.2 类 EntityManager

由于实体一般通过动态生成，所以需要一个全局的实体管理类进行实体的宏观控制，该类应运而生。这是游戏的核心类之一，是一个单例模式实现的类。由于和实体密切相关，在上一节中也在一些地方提到了这个类。该类的主要结构如表 4.8 所示。

这其中，EntityManager 的核心函数是 Step()，这也是物理世界和绘图世界同步的关键。Step()的过程如图 4.8 所示。

4.6.3 类 Tank

类 Tank 用于表现游戏中的坦克，是 Entity 的子类，在 Entity 的基础上添加了游戏相关要素。该类主要结构如表 4.9 所示。

坦克在构造和销毁时均有自己的粒子效果，坦克的方向由枚举值 ENTITY_DIRECTION 确定，有效的方向为上下左右 4 个，该枚举其实代表着实体的方向。可通过辅助函数把坦克或实体的旋转角度作为输入从而得到其对应的 ENTITY_DIRECTION 值。

由于敌方坦克在受伤时会根据剩余生命值的百分比更改自己的纹理，而且在由玩家击破后会得到一定的奖励分数，所以可以通过子类化 Tank 类得到 Tank_Enemy 类实现这些功能。前一点通过重载 Hit()实现，后一点通过增加数据成员和该数据成员的 getter，setter 方法实现。

表4.8　EntityManager类的主要结构

类型	声明	作用
数据成员	std::set<Entity*> m_allent;	保存当前游戏运行过程中所有的实体
	b2World* m_world;	物理引擎Box2D的世界接口，用于创建和维护物理世界
	PhysicsDebugDraw* m_debugdraw;	物理世界的调试绘图输出
	bool m_drawdebugdata;	是否开启物理世界调试绘图
	ContactListener* m_contactlistener;	物理世界碰撞监听对象
	std::set<b2Body*> m_b2bodytodel;	待删除的刚体列表
成员函数	bool IsValidEntity(Entity* ent);	判断一个实体是否有效
	void RegisterEntity(Entity* ent);	注册实体
	void DestroyEntity(Entity* ent);	销毁实体，整个实体销毁过程从这里开始
	void DestroyAllEntity();	销毁所有实体
	void SetDebugDraw(bool isdraw);	设置是否开启物理世界绘图调试
	void InitWorld();	初始化物理世界，由于游戏中存在的用户输入控制实体特点，将世界设定为无重力状态更为合理
	b2World* GetWorld();	返回物理世界接口
	void Step(float dt, bool drawonly = false);	模拟物理世界，即计算一段时间内物理世界的变化，反映到每个刚体上，并最终同步到绘图世界

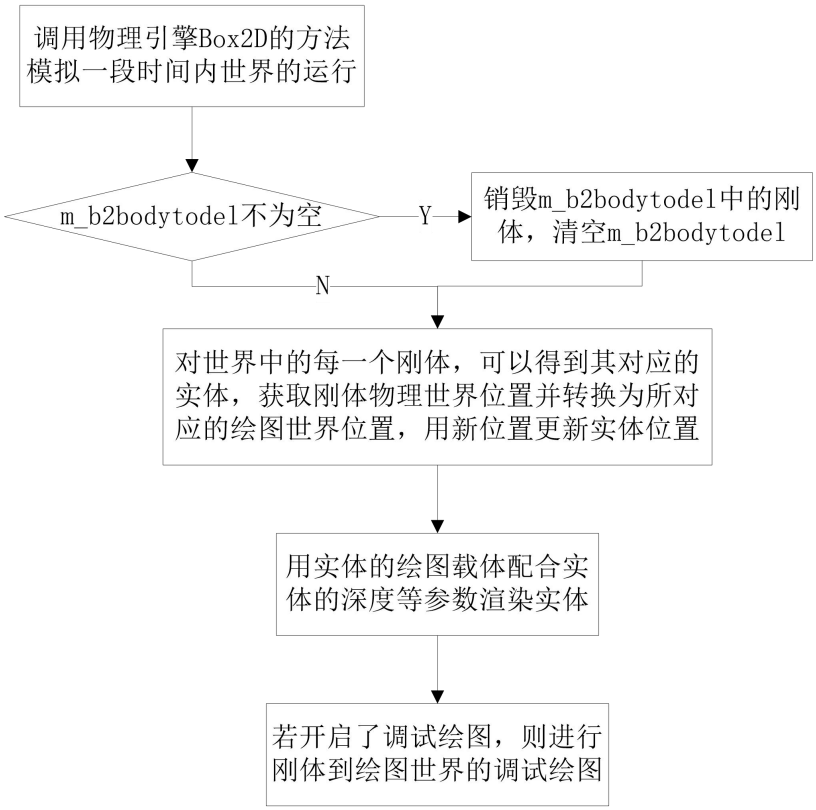

图 4.8　　Step()过程

表 4.9　　Tank 类的主要结构

类型	声明	作用
数据成员	int m_bulletpower;	表示子弹威力
	int m_teamid;	队伍标志,不同标志的坦克互相为敌，相同为盟友
	float m_speed;	坦克移动速度
成员函数	void Fire();	开火函数,调用后坦克即向当前方向开火,产生一个飞行的子弹实体
	void Move(ENTITY_DIRECTION dir);	移动函数,调用后坦克即向当前方向移动 m_speed 个像素
	void Move(ENTITY_DIRECTION dir, float speed);	移动函数的重载,可以自己指定移动距离

除以上功能外， Tank_Enemy 类在构造时还可以输入一个敌方类型参数，该参数用枚举值 TANK_ENEMY_TYPE 表示，总共有 4 种类型的敌方坦克，每种坦克的属性(如子弹威力，生命值，移动速度)都不一样，且每种坦克在受伤时更改的纹理也不同。

4.6.4 类 Bullet

该类用于表现游戏中的子弹，是 Entity 的子类。子弹是游戏中唯一拥有接触杀伤力的实体，子弹的构造只发生在坦克开火时，它是一类可穿透的实体。该类的主要数据成员如表 4.10 所示，成员函数则都是对数据成员的 getter 和 setter 方法。

表 4.10　　Bullet 类的主要数据成员

声明	作用
Tank* m_owner;	表示该子弹实体的拥有者，该值在坦克 Fire()的过程中被设定
int m_attack;	子弹的威力，该值应当和其拥有者的子弹威力相同，在坦克 Fire()的过程中被设定

子弹在其构造后便获得一个方向为子弹面朝方向的线速度，用于模拟子弹的飞行。子弹的初始位置由其拥有者设定，为了防止子弹一出生便和其拥有者发生碰撞，子弹的初始位置和其拥有者之间有一个很小的间隔。

4.6.5 类 Block

类 Block 表现游戏中的障碍物，是 Entity 的子类。游戏中的坦克在移动过程中遇到障碍物会停止其向当前方向移动，坦克可以通过开火发射子弹摧毁一些障碍物。Block 类是一类静态实体。

游戏中的障碍物可分成 2 种，可破坏和不可破坏的。可损坏的障碍物在受到损害时会改变自身的纹理，所以 Block 类需要重载 Hit()方法，这和 Tank_Enemy 类是类似的，Block 类在销毁时会有自己的粒子特效。

4.6.6 游戏奖励类

游戏奖励类表现的是游戏里的奖励元素。游戏中的奖励一旦和坦克接触便会发生某种效果，效果可以不仅限于作用在与奖励接触的坦克上。奖励每隔一段时间随机生成，若一段时间内没有坦克和奖励接触则奖励会消失。 游戏奖励类是可穿透实体，是处于无敌状态的。奖励在销毁时有自己的粒子特效。

Bouns_Base 是游戏奖励基类，该类是 Entity 的子类，所有的具体游戏奖励均是 Bouns_Base 的子类。该类定义了一个 OnHit()接口，用于和坦克发生碰撞时

调用，所有的子类需要实现该接口以实现不同的奖励效果。

游戏中的奖励有以下 4 种：

(1) Bouns_Power。坦克在拾取时可以提高自己的子弹威力。

(2) Bouns_Recover。坦克在拾取时可以恢复自身一定的生命值。

(3) Bouns_Invincible。坦克在拾取后的一段时间内可以使自己处于无敌状态。

(4) Bouns_PlusPoint。只有玩家控制的坦克拾取后才有效果，可以增加玩家一定量的分数。其他坦克拾取将不发生任何效果。

4.7 游戏状态类

使用游戏状态管理游戏流程十分方便，同时也提高了代码的可读性。它们也是游戏的核心类。每一个游戏状态都拥有自身的逻辑处理，图像渲染和输入控制，也可以轻松的进行状态的切换。游戏状态之间既相对独立又可以互相影响。

4.7.1 类 GameState

该类是游戏状态基类，定义了游戏状态驱动游戏流程所需要的接口，如表 4.11 所示。

表 4.11　　GameState 类定义的接口

接口声明	作用
virtual void Render() = 0;	用于处理该状态的渲染工作
virtual bool Think() = 0;	用于处理该状态的逻辑计算
virtual void OnEnterState();	当状态激活并运行时调用，可用来重置状态中各种参数设置

所有的状态子类需要实现 Render()和 Think()接口， OnEnterState()接口为可选，GameState 类为其实现了一个空的函数体。

4.7.2 类 GameStateManager

该类是游戏状态全局管理类，使用单例模式实现，有控制状态运行，更改当前状态等功能，其内部使用一个状态栈。该类的主要结构如表 4.12 所示。

该类的结构和使用非常简单，所有的状态转换都由该类负责进行，程序也可以在任意地点对当前状态进行改变。设置好状态栈后，程序只需要在主要桢调用和渲染调用运行 DoThink()和 DoRender()即可。

4.7.3 类 GameState_MainMenu

该类实现游戏的主菜单状态，是 GameState 的子类。该状态的界面由标题，菜单项和背景 3 个部分组成。主菜单状态时会播放主菜单主题的背景音乐。在该

状态中可以接受鼠标和键盘输入，用来响应菜单项。

表 4.12　GameStateManager 类的结构

类型	声明	作用
数据成员	stack<GameState*> m_allgs;	内部状态栈,用于保存和维护状态
成员函数	GameState* GetCurState();	返回当前状态（栈顶状态）
	void ChangeState(GameState* gs);	改变当前状态
	void PushState(GameState* gs);	向状态栈推入一个状态
	void PopState();	从状态栈推出一个状态
	bool DoThink();	调用当前状态的 Think()
	void DoRender();	调用当前状态的 Render()

界面的标题和菜单项使用了 HGE 游戏引擎简单的 GUI 功能部分，分别是文本 GUI 对象和菜单项 GUI 对象，它们被添加到同一个 hgeGUI 对象中，该对象可以同时管理多个子 GUI 对象，使它们的逻辑和渲染同步。当菜单项进入屏幕时会按顺序从上往下以少许延迟进入，产生一个动画效果；鼠标与菜单项接触时会产生音效，菜单项会高亮并表现出被选中的样子；可以通过键盘上的上下箭头键来选择不同的菜单项，同样有选中效果；当选择某一菜单项后且需要退出菜单时菜单项同样会按顺序以少许延迟退出，产生动画效果。

为了增加界面的丰富性，界面的背景将按一定速率和角度重复滚动显示，为菜单界面增加动感。

主菜单中包含 Play Solo，Options 和 Exit 这 3 个菜单项，选择 Play Solo 后游戏将转入游戏中状态，选择 Options 后游戏转入设置菜单状态，选择 Exit 游戏将正常退出，Esc 键也代表着选择 Exit。

4.7.4 类 GameState_Option

该类实现游戏的设置菜单状态，是 GameState 的子类。该状态的界面由标题，设置项和背景组成。该状态的背景音乐和主菜单状态一致。

界面标题和设置项和主菜单的组成框架基本相同，背景也是和主菜单一样的重复滚动背景。这里的设置项只包含背景音乐的大小调节，玩家可通过键盘的上下键进行音量调节。按下 Esc 键游戏状态将返回主菜单状态。

4.7.5 类 GameState_Play

该类实现游戏的游戏中状态，可以说是整个游戏最重要的流程部分。该状态实现读取游戏关卡，载入游戏元素，提供游戏控制和胜负判定等功能。该类的主要结构如表 4.13 所示。

表 4.13　GameState_Play 类的主要结构

类型	声明	作用
数据成员	string m_curlevel;	当前关卡文件名
	string m_nextlevel;	下一关卡文件名
	string m_levelname;	当前关卡名
	Tank* m_player;	玩家的坦克实体
	std::vector<Block*> m_blocks;	当前关卡的障碍物集合
	std::vector<EnemyTankInfo> m_enemyinfo;	所有敌方坦克出生点的信息存储集合
	Time_Cycle* m_enemyspawntimer;	敌方出生计时器
	std::vector<AI_Base*> m_ai;	所有 AI 集合
	Time_Cycle* m_bounstimer;	奖励出生计时器
	Bouns_Base* m_bouns;	游戏中的随机奖励，同一时刻只有一个奖励出现
	GAMESTATE_PLAY_STATUS m_gamestatus;	当前游戏进行状态（进行中，胜利，失败等）
	bool ResetGame(bool curlevel = false);	重置游戏流程
	void AddSecTimerCallBack(gsplaysectimercallback callback, float t, void* data = nullptr, bool oncleancall = false);	添加秒计时器回调
	void OnEntityContact(Vec2b2fixture vecb2fix);	游戏碰撞回调处理

下面将对几个游戏中状态的重要结构进行详细阐述。

(1) 游戏的重置

游戏的重置由成员函数 ResetGame()实现，游戏重置包括了清理游戏元素，载入关卡，载入游戏元素等步骤。该函数有表示是否重置为当前关卡的 curlevel

参数。游戏重置的流程如图 4.9 所示。

(2) 游戏逻辑判断处理

该状态的游戏逻辑判断处理集中在 Think()里，包括处理按键输入，准备要显示的信息字符串，出生敌方坦克，更新奖励等过程。游戏中敌人的出生和奖励的更新分别由一个计数器控制，当计数器到达周期后方可执行相应功能。游戏逻辑判断流程如图 4.10 所示。

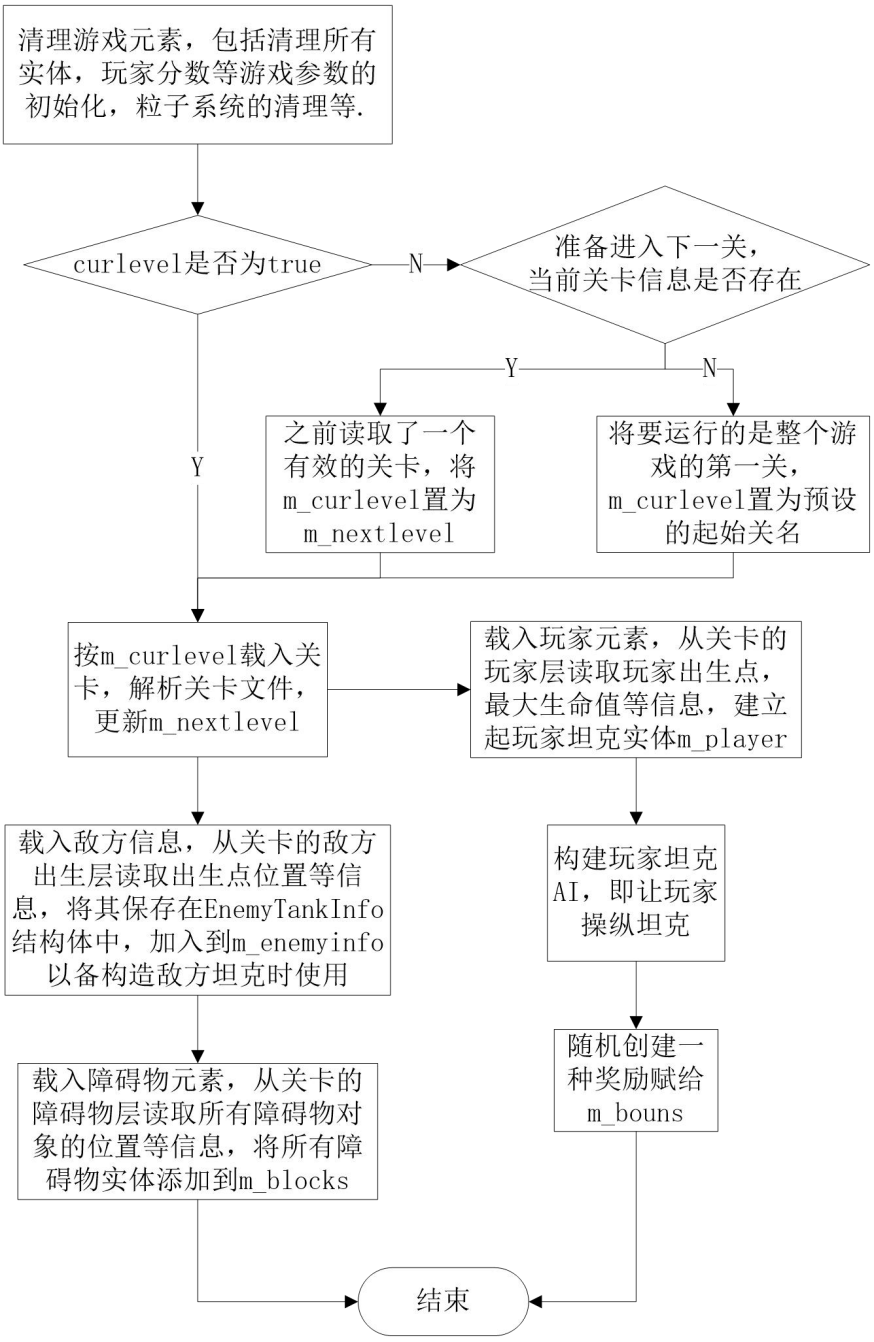

图 4.9　　游戏重置流程图

(3) 游戏碰撞处理

碰撞处理通过 OnEntityContact()进行，主要处理 AI，奖励和子弹的碰撞，其中子弹的碰撞最为复杂。当有实体被销毁时则不继续进行碰撞处理。AI 和奖励的碰撞处理流程如图 4.11 所示。

子弹的碰撞检测流程较长且含有大量的判断，使用流程图将过于复杂且难以理解，故采用文字表述重点的方法进行描述。其流程的重点如下：

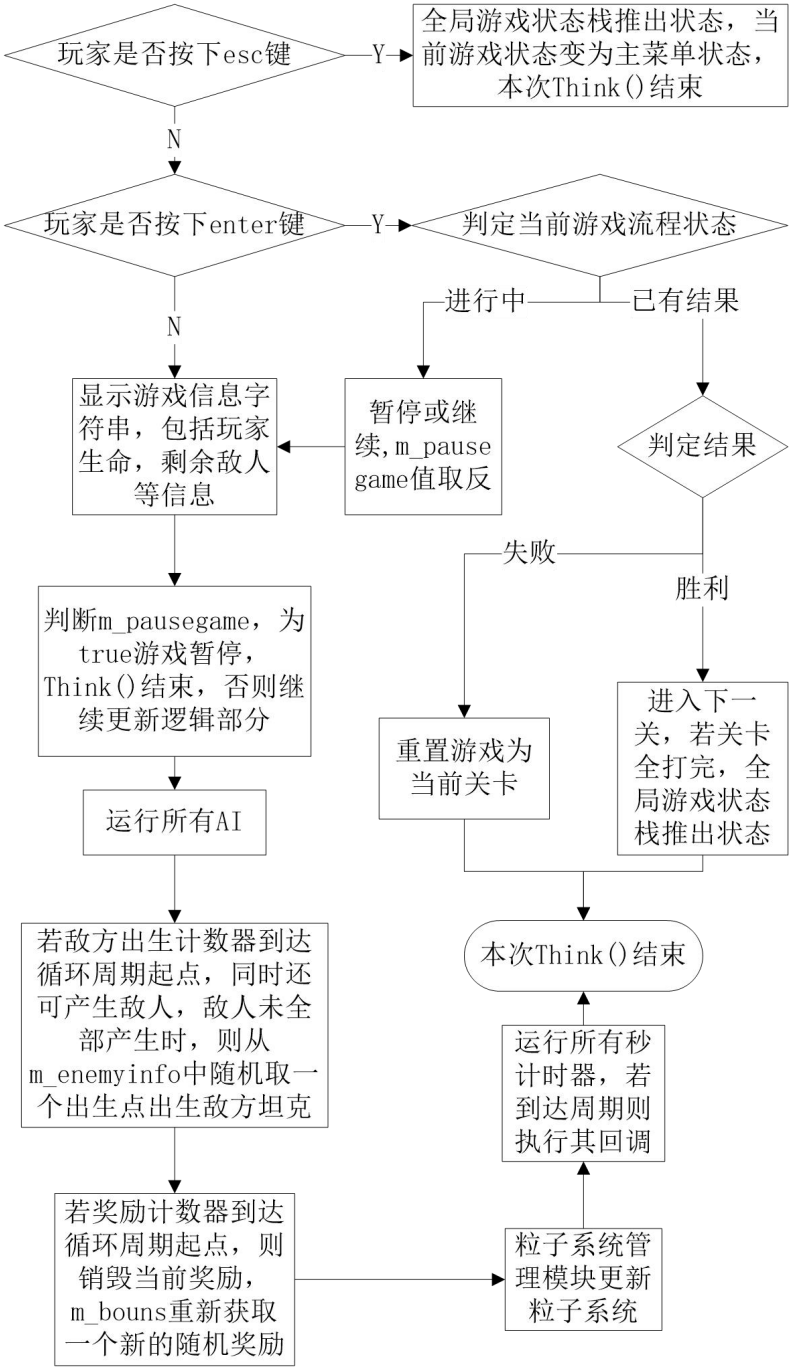

图 4.10　　游戏逻辑判断流程

1. 当子弹和同为盟友的坦克碰撞时，什么也不做。

2. 否则子弹将对碰撞对象造成伤害(调用对方的 Hit())。

3. 若子弹没把对方消灭，根据对方种类的不同(可摧毁 Block，不可摧毁 Block，Tank)播放不同的音效。

4. 若子弹消灭了对方，如果对方是 Tank，从 m_ai 中销毁控制该 Tank 的 AI；若是玩家被消灭，则将 m_player 无效化，将游戏流程状态设为失败，播放失败音效，否则从 m_enemy 里删除被消灭的敌方坦克信息，若是玩家打死敌方则玩家增加杀死该种坦克可得到的分数。无论怎样，只要 Tank 被消灭则播放 Tank 摧毁的音效。

5. 若子弹消灭的是 Block，从 m_blocks 里删除该 Block，如果是玩家消灭的，则可以获得消灭 Block 的加分，Block 被消灭则播放 Block 摧毁的音效。

6. 无论被消灭的是何种实体，都将通过 EntityManager 的 DestroyEntity 销毁该实体。

7. 如果与子弹碰撞的不是奖励实体，则显示子弹碰撞的粒子特效，同时摧毁子弹实体。

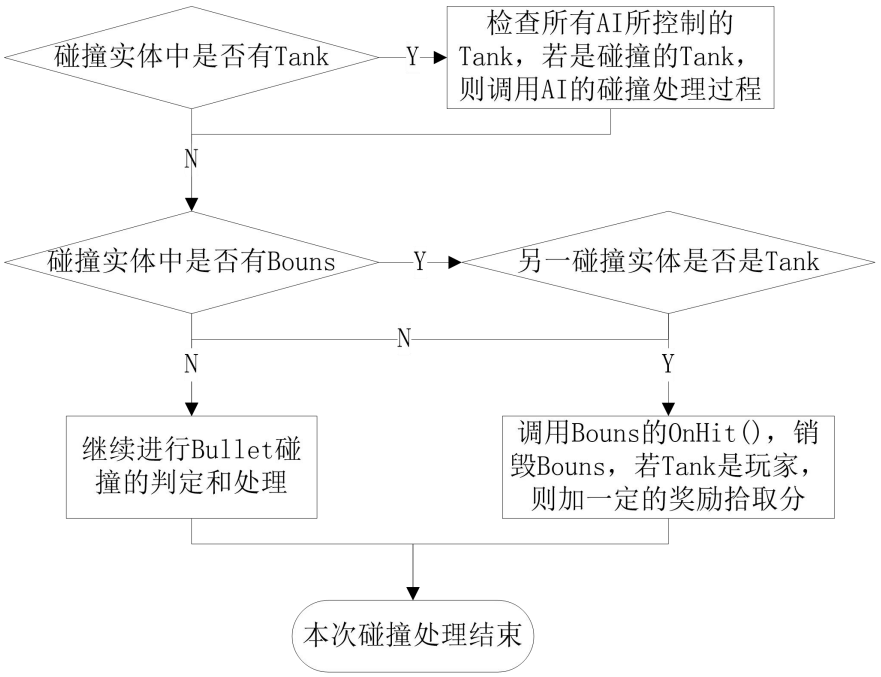

图 4.11　　AI 和奖励碰撞处理流程

(4) 秒计时器系统

该状态提供一个秒计时回调的功能，通过特定的一段时间后可以调用一个指定的函数。该功能可用于实现有限时的奖励效果。

若要使用该功能，用户需要调用 AddSecTimerCallBack()并提供时间，回调

函数等参数，还可以提供一个自定义数据指针指向所要的数据，这类的信息将被整合到 SecTimerData 结构体中，其中时间参数会被实现为一个秒计数器，该结构体会保存到 m_sectimer 集合中。在每次的 Think()的最后将会遍历 m_sectimer，对于已经到达循环周期的秒计数器，将会执行其回调函数并将其 SecTimerData 结构体移出 m_sectimer。

4.8 人工智能类

游戏中的敌方坦克需要定义一定的 AI(人工智能)以让其自动运作，AI 用来控制坦克的移动和开火策略。同样，玩家对于坦克的控制也可以抽象为一种 AI 模型。AI 的运行通常是在每一桢中调用其运行接口来进行。

4.8.1 类 AI_Base

该类是所有 AI 的基类，定义了 AI 运行所需的数据成员和 AI 运行的接口。该类的结构如表 4.14 所示：

表 4.14　　AI_Base 类的结构

类型	声明	作用
数据成员	Tank* m_tank;	指向该 AI 所控制的 Tank
成员函数	Tank* GetTank();	返回该 AI 控制的 Tank
	virtual void Think() = 0;	AI 的运行接口，所有 AI 的行为都发生在这里
	virtual void OnContact(Entity* ent);	当 AI 和其他实体碰撞时的处理函数

其中的核心是 Think()，该函数在每一桢中都会调用，用来模拟 AI 的连续思考。

4.8.2 类 AI_Player

该类是 AI_Base 类的子类，将玩家的控制抽象为一种 AI 模型。该类中包含一个秒计时器，用来限定玩家操纵坦克开火的频率。

该类的 Think()十分简单，检查玩家是否按下以执行特定功能。WASD 这 4 个键分别表示玩家控制坦克向上，左，下，右移动，Space 键则表示玩家控制坦克开火，开火时将有开火音效。

4.8.3 类 AI_Random

该类是 AI_Base 类的子类，表示使用随机驱动的 AI 模型。类中存在 2 个计时器，一个和 AI_Player 类一样，用于限定随机 AI 操纵坦克开火的频率，另一

个用于控制 AI 为坦克选择移动策略的频率。

该类的 Think()分为移动行为控制和开火行为控制。在移动行为控制中如果移动策略计时器到达循环周期起点则让坦克更改一次移动方向，这个周期设为 20 桢；开火行为控制中，AI 有 1/30 的几率决定是否开火，在决定开火并且开火计时器到达循环周期起点时才会真正的开火，该计时器周期为 0.2 秒。

4.9 辅助类和辅助函数

除了以上介绍的组成游戏主体部分的类和模块外，本项目还定义了一些其他的辅助类和函数满足一些较为基础的需求。

4.9.1 计时/计数类

主要提供循环/非循环计时/计数的功能，该类系的基类为 Base_Cycle，只定义了 ReachCycleStart()接口，当此接口返回 true 时表示计数器到达了一个周期的起点，在周期中将返回 false。

从 Base_Cycle 子类化的类还包括类 Time_Cycle(以调用函数次数为周期计数)和类 Second_Cycle(以秒为周期计数)，这 2 种计数器类只有次数的单位定义不同，完成的功能是一样的。

4.9.2 辅助函数

辅助性函数的功能较杂，表 4.15 简单介绍了部分辅助函数：

表 4.15　　部分辅助函数简介

函数	作用
ENTITY_DIRECTION GetEntityDirection(float rot);	通过角度计算实体的方向枚举值
ENTITY_DIRECTION GetDirBetweenEnt(Entity* ent1, Entity* ent2);	判断 ent2 在 ent1 的什么方向
int Randint(int a, int b);	返回[a,b]范围内的随机整数
ENTITY_DIRECTION RandDir();	返回随机方向
bool HitChance(int a, int b);	该函数有 a/b 的几率返回 true
bool EntTypeHas(char* type, Entity* ent1, Entity* ent2, bool& asc);	某一个 ent 的类型是 type 是返回 true,同时使用参数 asc 确定 ent1 和 ent2 哪一个的类型是 type

4.10 界面展示与分析

游戏代码基本完成后，经过调试，排错，优化，现在的坦克大战游戏已经可以正常的运行。游戏可以准确的提供主菜单，设置菜单和游戏中这 3 个状态的全部功能，如图 4.12、图 4.13、图 4.14、图 4.15、图 4.16、图 4.17 所示为整个游戏过程中的部分状态或界面截图。

图 4.12　　主菜单界面

图 4.13　　设置界面

图 4.14　　游戏开始，初始关卡界面

图 4.15　　关卡胜利，可进入下一关

图 4.16　　关卡失败，可重新进行挑战

图 4.17　　丰富的敌方坦克种类

由截图可知所有的功能都能够正常使用，没有出现明显的错误和漏洞，但是游戏在一些细节方面仍然需要完善，比如游戏 GUI 只提供了最基础的功能，不够美观；游戏没有地形因素，障碍物种类也较少；敌方 AI 智力偏低，只能靠数量提高游戏难度等。

在完成了本次设计的整体框架后，我们便可以自行对坦克类型，奖励，AI 等因素进行扩展，以进一步增加游戏的趣味性。总的来说本次设计在功能上已经基本达到要求，以上提到的细节有待日后改善。

4.11 本章小结

本章在第三章的理论基础上对坦克大战游戏进行了模块划分和具体代码的实现，建立了一个较为清晰的游戏框架体系，详细解释了各个核心类的构成，如 GamePlay 类，GameState 类体系和 Entity 类体系，同时对一些重要的流程也做了重点讲解，如实体的产生和销毁，游戏的重置和游戏碰撞的处理，在最后还对成果进行了截图展示。本章能够较为完整的体现整个项目的规模和概况。