实体和组件

与任何软件一样，设计一个复杂的游戏需要规划一个好的架构。随着游戏变得更大、内容丰富，构建简单游戏演示时，一些自然产生的设计决定可能会成为障碍。GameplayKit提供了一个架构，帮助您从一开始就计划更好的可重用性，将与游戏不同方面相关的担忧分开。

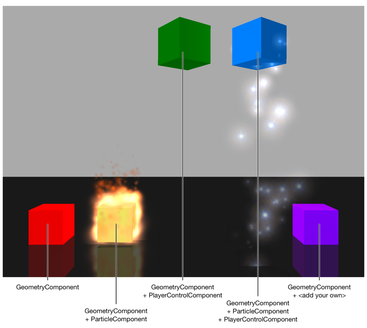
在这个架构中，*实体*是与游戏相关的对象的通用类型。实体可以表示对游戏至关重要的对象，例如玩家和敌人角色，或者仅仅存在于游戏世界中而不与玩家互动的对象。例如关卡中的动画旋转。实体甚至可以是您游戏的概念或用户界面元素，例如决定何时何地向关卡添加新敌人角色的系统，或管理玩家角色穿戴的设备的系统。

实体通过成为*组件*的容器来获得其功能。组件是处理任何实体外观或行为的特定、有限方面的对象。由于组件的功能范围有限，并且不绑定到特定实体，因此您可以在多个实体之间重用相同的组件。

示例：盒子

要快速查看实体组件架构，请下载示例代码项目*Boxes: GameplayKit实体组件基础知识*。Boxes游戏中的每个盒子实体由一组组件定义，如图3-1所示。单个组件不知道彼此以及它们所属的实体，因此您可以轻松地添加新实体和新行为，而无需在代码中进行复杂的相互依存关系。

图3-1 盒子样本代码项目在行动



使用实体和组件进行设计

实体组件设计模式是一种偏向于组合而不是继承的架构。为了说明基于继承和基于构图的架构之间的区别，请考虑如何设计一个具有以下功能的"塔防"风格游戏示例：

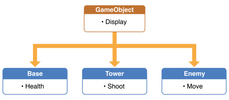
- 玩家在游戏场地的一侧有一个"基地"位置。
- 敌人角色周期性地在游戏场的另一边成波逐流地出现，向基地攻击它。如果太多的敌人到达基地，游戏就结束了。
- 为了防止敌人攻击基地，玩家在游戏领域放置塔楼。这些防御阻塔会自动向敌人开火。

这种设计使用基于继承的设计很容易建模：

- `Base`类可以建模Base，除了保持其显示表示（如SpriteKit游戏中的`SKSpriteNode`对象）和跟踪它持续了多少敌人的攻击外，该基地类几乎没有其他事情要做。
- An `Enemy` class could provide a display representation and logic to handle movement.
- `Tower`类可以提供显示表示和逻辑，以处理对敌人的瞄准和射击。

所有这些类都需要处理可视化表示，以便功能可以在常见的`GameObject`超类中实现。从而产生如图3-2所示的类层次结构。

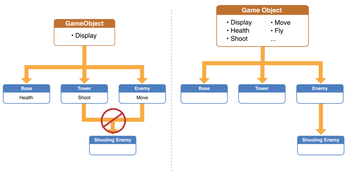
图3-2 基于继承的游戏设计



基于继承的架构阻碍了游戏设计演变

As you refine your game, you might choose to add more kinds of enemies and towers or add new capabilities to your game objects. For example, you might add enemies that fire back at the towers. However, this presents a problem—the new `ShootingEnemy` class can't reuse the targeting and firing code from the `Tower` class because it doesn't inherit from that class. To attempt to solve this problem, you might move that code into the `GameObject` class so that you can use it in both the `Tower` and `ShootingEnemy` classes.

图3-3 基于继承的设计发展问题



然而，如果您的游戏设计继续沿着这些方向发展，最终一个根类几乎拥有每个子类的所有功能，而每个子类提供的功能很少。这种情况导致代码复杂且难以维护，因为根类实现在做任何事情之前需要检查它使用的子类标识，并且必须在根类的各种条件行为中仔细地适应新功能。

基于构图的架构让进化的游戏设计变得简单

实体组件模式鼓励您思考每个游戏对象*的作用*，而不是根据每个游戏对象来组织游戏对象模型。在本示例中，您可以为基地、敌人或塔楼可能预期的每个功能位制作组件。一个组件可能处理视觉表示；另一个组件可能处理目标定位和发射；另一个组件可能处理移动；另一个组件可能跟踪遭受的损坏程度，并相应地更新用户界面元素；依此类推。为每个独特的功能领域创建一个新的`GKComponent`子类。

然后，不要为每种游戏对象创建类，而是使用`GKEntity`类作为组件的通用容器。例如（如图3-4所示），一个`GKEntity`实例可以表示一个塔，因此它包含视觉和发射组件。另一个`GKEntity`实例可以代表敌人，因此它包含视觉和运动组件。要添加可以射击的敌人，只需将移动和射击组件同时添加到同一实体。

图3-4 实体-组件设计组成功能

使用实体和组件构建游戏

实体组件设计模式本身就是组织代码的一种方式。GameplayKit扩展了这种模式，以纳入游戏开发的另一个常见概念——定期更新。结合这些概念为构建游戏提供了一个有用的架构。

定期更新的驱动游戏

游戏引擎（如SpriteKit和SceneKit，但也包括直接使用Metal或OpenGL构建的自定义引擎）通常在称为*更新/渲染周期*的紧密循环中运行与游戏相关的代码。在更新阶段，游戏更新其所有与游戏相关的内部状态，执行操作，例如轮询控制器输入，模拟敌人角色入工智能，安排要播放的动画，以及调整屏幕上代表游戏角色和用户界面的元素。在渲染阶段，游戏引擎处理动画和物理等自动化功能，然后根据当前状态绘制场景进行显示。通常，游戏引擎的设计使游戏程序员只控制更新逻辑，而引擎本身负责渲染阶段——SceneKit和SpriteKit就是这种情况。

The `GKEntity` and `GKComponent` classes in GameplayKit make use of this notion. Each time your game's update handler executes (for example, the `update:` method in a SpriteKit scene or the `renderer:updateAtTime:` method in a SceneKit renderer delegate), you can dispatch to the `updateWithDeltaTime:` methods of the components that provide your game logic. GameplayKit provides two ways to handle this dispatch:

- **每个实体更新。**在实体很少的游戏中，您可以迭代游戏中活动实体的列表，并调用每个实体的`updateWithDeltaTime:`方法。然后，`GKEntity`类将更新消息转发到其附加的每个组件。
- **Per-component updates.** In a more complex game, it can be useful to ensure that all components used by all entities update in a strictly defined order, without needing to keep track of which entities have which components. In this case, you can create `GKComponentSystem` objects, each of which manages all components of a specific class. Then, call the `updateWithDeltaTime:` method for each component system in your game, and the component system forwards update messages to all components of its class, regardless of which entities those components are attached to. For details, see *GKComponentSystem Class Reference*.

实体组件设计游戏实例

迷宫示例代码项目对几个经典街机游戏进行了变体。在游戏中，玩家角色必须导航迷宫。然而，四个敌人角色追逐玩家，与敌人接触将玩家角色重置为初始初始位置。

> 注
> 本节讨论示例代码项目迷宫：GameplayKit入门的功能。下载它，在Xcode中跟随。

该项目展示了实体组件设计（还展示了其他几个GameplayKit功能），在游戏中，玩家角色和敌人角色都由实体表示，每种类型角色使用不同的组件集。在示例代码项目中，您将找到三个`GKComponent`子类：

- `AAPLSpriteComponent`：该类为每个实体管理将游戏逻辑转换为屏幕上的动作。其他组件将迷宫建模为整数网格，并调用实体的精灵组件将实体的网格位置与提供该实体视觉表示的SpriteKit节点同步——在此过程中动画该表示形式。玩家角色实体和敌方角色实体都使用此组件。

  由于精灵组件处理所有与精灵工具包相关的逻辑，用于处理游戏角色，其他组件不需要了解场景维度或动画。如果游戏视觉设计在未来发生变化（甚至切换到不同的渲染引擎，如SceneKit），其他组件不需要重写。
- `AAPLIntelligenceComponent`：该类仅供敌方角色实体使用，并提供允许他们独立移动和对玩家行动做出反应的逻辑。为了跟踪敌人在任何给定时间应该做什么，情报组件使用状态机（请参阅状态机一章）。部分敌人逻辑还使用视则系统（见规则系统一章）来随意可能导致不同行为的条件。该组件还使用寻路图（请参阅寻路章节）将敌人路由到迷宫中，并调用精灵组件处理该动作的屏幕表示。

  由于智能组件是广义的，它可以被重用到其他类型的实体。例如，游戏的未来变体可能包括计算机控制的盟友角色来追逐敌人。
- `AAPLPlayerControlComponent`：该类仅供玩家角色实体使用，将定向输入（可能来自游戏控制器、键盘或触摸屏事件）转换为游戏世界中玩家角色的运动，然后调用精灵组件在屏幕上动画该运动。

  由于控制组件在迷宫中介于所需方向和运动之间，因此它可以被重用用于其他类型的实体。例如，游戏的未来变体可以通过为每个玩家使用单独的`AAPLPlayerControlComponent`实例来增加多人游戏支持。

虽然可以直接使用`GKEntity`类（作为自定义`GKComponent`子类的容器），但通常情况下，多个组件需要共享有关其行为的实体实例的信息。您可以创建一个`GKEntity`子类，而不是将此类信息保存在组件类中（并且必须在多个组件之间同步）。在迷宫游戏中，所有组件类都需要知道它们所控制的实体在代表迷宫的整数网格上的当前位置。因此`AAPLEntity`类存储这些信息。