

Developing a Java Game from Scratch

刘佳璇^{1*}

1. 南京大学计算机科学与技术系

* 通信作者. E-mail: 13573529579@163.com

1 引言

在传统的 Java 教学中,语言特性、库使用与面向对象思想往往被割裂开来讲解,学生较少有机会在一个长期、复杂且具有明确工程目标的项目中,系统性地运用这些知识。《高级 Java 程序设计》课程通过分阶段布置游戏开发作业,引导学生从零开始构建一个可演进的软件系统,为理解 Java 在实际工程中的能力边界提供了良好的实践环境。

本文所描述的项目正是在该课程框架下完成的一个综合性游戏开发实践。该项目的核心目标是通过游戏这一高度综合的应用场景,系统性地实践面向对象设计、并发编程、网络通信、渲染抽象以及软件工程方法,并在持续迭代中总结可复用的工程经验。

项目大体经历了五个阶段:j03 以 Swing 为起点搭建最小可运行引擎;j04 在对象数量上升后进行性能剖析并引入并行计算;j05 迁移到 LWJGL/OpenGL,建立渲染后端抽象并实现录制回放;j06 基于 NIO 添加网络联机与客户端插值;最终在 j07 形成完整交付形态,引入 Maven、单元测试与 CI,并在服务端加入单机模式与存档功能,实现“网络对战/单机游玩/回放复盘”三类体验的统一。

2 开发目标与游戏设计构想

2.1 总体目标

本项目的目标是开发一个完整的 Java 游戏系统,它既能作为课程作业的可运行作品,也能作为一个“可复用、可扩展”的轻量级游戏引擎骨架,支持后续继续添加玩法与内容。综合课程要求与个人目标,我将目标拆解为三个层次:

- **可运行**: 具备稳定的游戏循环、输入、更新与渲染能力,能够在不同机器上重复运行且行为一致;
- **可扩展**: 采用清晰的模块边界与抽象接口,新增对象类型、系统或玩法时尽量做到“增量开发”,避免牵一发而动全身;
- **可交付**: 提供单机与联机两种运行模式,并具备录制、回放与存档等完整产品化要素,同时满足 Maven 构建、单元测试与 CI 的工程要求。

因此,本项目的成果不仅是一个可运行的游戏程序,更是一个具备基本通用性的轻量级游戏引擎。我在实现过程中保留了接口层与扩展点,使其能够支撑后续更多拓展(例如加入更多渲染特效、复杂 AI、战斗技能体系、地图编辑等)。

引用格式: 刘佳璇.

. 中国科学: 信息科学, 在审文章

Liu J. Developing a Java Game from Scratch. Sci Sin Inform, for review

2.2 游戏形态与灵感来源

在具体游戏形态上,项目选择了结构相对简单但技术要素齐全的实时射击类玩法:玩家通过键盘控制角色在二维平面内移动,系统周期性生成敌对单位,玩家需要在有限空间内躲避、反击并尽可能存活更长时间。其本质是一个“实时更新 + 状态同步”的系统,非常适合用来承载网络联机、回放与存档等非功能需求。

这一设计思路的灵感主要来源于经典的 2D 射击类游戏。这类游戏对实时性要求较高,同时又不依赖复杂的资源系统,能在较小的内容规模下逼迫开发者直面工程问题:例如对象生命周期管理、碰撞检测的复杂度、输入事件的去抖与一致性、渲染抽象与性能边界、联机状态同步的策略选择等。换言之,该玩法可以让时间投入更多地集中在架构与质量上。

2.3 玩法与交互的具体设定

为了让工程结构与玩法逻辑解耦,项目在“规则层”只保留少量、清晰且可扩展的核心设定:

- **对象类型:** 玩家、敌人、子弹等,均以统一的对象模型表达;
- **胜负与反馈:** 以生存时长与击杀数作为主要指标,碰撞敌人或子弹则判负;同时以 HUD 文本展示当前状态,保证“可观察性”;
- **模式切换:** 单机模式与联机模式共享同一套场景与对象建造逻辑,只在输入来源与状态权威来源上不同。

上述设定体现了工程可控性:规则越“明确且稳定”,引擎越容易验证与测试;而当玩法需要扩展时,只需在规则层增加新的系统或组件即可,无需破坏底层框架。

3 总体架构设计与设计理念

3.1 总体结构

项目整体采用高度模块化的设计。其包结构与物理目录结构保持一致,核心模块包括:

- **core:** 游戏引擎主流程、主循环与全局调度;
- **scene:** 场景生命周期与切换管理;
- **objects / components:** 基于 ECS 的实体与组件;
- **graphics:** 渲染抽象与具体实现 (Swing 或 LWJGL/OpenGL);
- **input:** 输入采样、事件去抖与动作映射;
- **net:** 网络通信、协议编解码、缓冲与同步策略;
- **recording / replay / save:** 录制、回放与存档系统。

我在依赖关系上使用“由内向外”的单向依赖:core/scene/objects 作为内核层,不直接依赖具体渲染实现或网络实现;graphics 与 net 作为外设层,通过接口注入到内核,从而保证替换渲染后端或通信机制时,上层玩法逻辑不需要大范围改动。该策略在 j05 从 Swing 迁移到 LWJGL/OpenGL 时尤其关键:如果游戏对象直接调用 Swing 的绘制 API,迁移几乎等同于推倒重来;而在渲染接口存在的前提下,迁移工作主要集中在渲染后端与资源管理。

3.2 ECS 架构的引入

项目核心采用 Entity-Component-System (ECS) 架构,将传统面向对象中“继承层级复杂、职责混杂”的问题拆分为:

- **Entity:** 仅作为唯一标识的容器,负责聚合组件;

- **Component**: 描述数据或单一职责行为的可组合单元, 例如位置、速度、渲染外观、碰撞体等;
- **System**: 对“拥有某类组件集合”的实体批量执行过程逻辑, 例如物理更新系统、碰撞系统、渲染系统、网络同步系统等。

这种设计显著降低了类之间的耦合度: 新增一个“能力”通常意味着新增一个组件与对应的系统, 而不是在某个庞大的继承树上添加子类并重写方法。更重要的是, ECS 天然适配并行化与数据局部性优化: 系统可以以“组件列表”为主要遍历对象, 避免对大量对象进行动态派发; 并且当需要并行计算时, 系统可以按批次切分组件集合, 降低共享状态带来的竞争。

3.3 核心抽象与职责边界

我将核心抽象划分为三层:

- **GameEngine**: 只负责主循环、时间步推进、窗口生命周期与全局模式切换;
- **Scene**: 负责一组对象的生命周期 (`initialize/update/render/clear`) 以及场景切换时的资源释放;
- **GameObject**: 作为实体容器, 提供组件增删查与局部渲染委托; 其本身尽量不承载复杂业务逻辑。

在该结构下, “玩法”被放置在场景或系统中, 而不是散落在对象类里。实践证明, 这种职责拆分使得网络模式、回放模式与单机模式之间可以共享同一套对象构造与系统流程, 只需替换输入源与状态权威来源即可。回放场景 (`ReplayScene`) 就是一个典型例子: 它并不需要复刻完整的游戏 AI 和逻辑, 而是通过读取关键帧并插值渲染, 实现了接近实时运行的视觉复现。

3.4 设计理念: 可演进与可验证

回顾整个项目, 我认为最重要的设计理念是“可演进”和“可验证”。

所谓可演进, 是指在需求持续变化、功能持续增加时, 代码结构仍能够承受增量开发, 而不是在某个阶段失控。为此我刻意做到三点: 第一, 模块之间通过接口交互; 第二, 数据结构稳定且可序列化 (为回放、存档、网络同步服务); 第三, 避免在多个地方重复表达同一份规则 (例如对象外观与创建逻辑统一由 `EntityFactory` 管理)。

所谓可验证, 是指系统的核心行为可以被观察、复现与测试。游戏项目天然存在随机性与非确定性, 若不控制, 其调试与测试成本会急剧上升。为此我在关键路径上尽量引入“确定性约束”: 例如固定时间步的逻辑更新、输入事件的离散化 (`just pressed` 语义)、关键帧时间戳的统一、以及回放/网络共用同一套解析工具。这些选择看似牺牲了一部分灵活性, 但换来了可维护性与可测性。

4 关键技术问题与解决方案

4.1 游戏循环与更新节奏控制

引擎的主循环将“规则推进”与“画面呈现”分离: 规则层以稳定的时间步推进世界状态 (对象位置、技能判定、碰撞与结算等), 渲染层则尽可能按照当前窗口刷新节奏将最近一次可用的状态呈现出来。这样做的收益在于: 当渲染短暂卡顿时, 规则推进仍可按照既定步长补齐, 避免“慢机少算几帧导致游戏行为不同”的问题; 同时也为网络同步、回放插值提供统一的时间基准。

在实现组织上, 我把时间相关的概念拆成三类: **采样时间** (来自系统时钟, 用于估算真实经过时间)、**逻辑时间** (用于规则更新的离散 `tick/step`)、**展示时间** (用于渲染插值或平滑)。无论状态来自本地计算、网络

同步还是回放文件，最终都以“时间戳/逻辑步 + 状态快照”的形式进入呈现层，从而保证数据流一致、接口稳定。

4.2 并行计算与并发控制

随着场景中对象数量增多，更新管线里最容易出现 CPU 压力的环节通常是：批量对象的行为更新、碰撞检测与结算、以及网络/录制等与主线程并行的 I/O 工作。为降低主循环抖动，我在设计上遵循两条并发原则：

第一，**主线程保持权威**：与“世界状态一致性”强相关的写操作（生成/销毁对象、血量与计分结算、场景切换）尽量集中在同一线程中完成，避免多线程同时写同一份世界数据带来的竞态与不可复现 Bug。

第二，**并行只做纯计算或可合并的工作**：对于天然可并行、且可以在帧末合并结果的环节（例如对对象集合做分段计算、预计算潜在碰撞对、或把 I/O 解析成中间事件），可以使用 Java 标准并发工具（线程池、任务队列等）把计算拆分出去；主线程在一个确定的同步点收集结果，再统一落盘/入队/应用到世界状态。即便最终版本中移除了专门的性能基入口代码，这种“把可并行阶段变成可控任务”的结构仍能在对象规模上升时提供更稳定的帧时间分布。

4.3 通信效率：协议设计、包体控制与背压

本项目在 `net` 模块中实现了联机模式。由于游戏属于“强交互、状态变化频繁”的类型，若直接把完整世界状态逐帧发送，会导致包体膨胀、带宽浪费与延迟上升。为此我在通信层做了三类约束：

其一，**区分输入与状态**。客户端更适合上报“输入/意图”（例如方向、技能触发、关键操作），由服务端推进规则并生成权威状态；服务端对外广播的则是“状态快照/增量”，使客户端能够在不完全一致的渲染节奏下复现同一局游戏进程。这样做能显著降低客户端侧的复杂度：客户端不需要承担全部规则一致性责任，联机时更像“可视化终端 + 输入采集器”。

其二，**控制包体结构与频率**。在序列化层面，消息以结构化字段组织，并避免把高频但低价值的数据重复发送（例如可由客户端本地推导的展示信息）。在发送频率上，不强制与渲染帧率绑定，而是以逻辑步或固定网络 tick 为准，降低“高帧率机器发送更多包”的不公平现象。

其三，**引入背压与容错**。当网络波动导致消息积压时，与其无上限排队，不如丢弃过旧的可替代状态、保留关键输入或最近快照，保证“最新状态优先”。这一策略与回放/录制的“按时间步存储快照”天然兼容：网络侧只要保证关键帧/关键输入不丢，视觉层就能通过插值与平滑掩盖短期抖动。

4.4 客户端插值：在抖动下保持视觉平滑

联机模式下，客户端从网络接收的状态往往是离散且不均匀的：延迟、抖动会造成状态到达间隔变化，若直接把最新状态“生硬应用”，画面会出现跳变。为提升观感，我在客户端采用“延迟一小段展示时间 + 插值/平滑”的思路：渲染层不必追求显示到绝对最新的状态，而是保持一个很小的缓冲窗口，在窗口内对位置、朝向等连续量做插值或平滑过渡。

这里的关键在于：**插值只影响展示，不反向修改规则状态**。规则层仍以服务端权威的逻辑步为准；插值层只是把两个相邻快照映射成连续的画面，从而在网络抖动时维持可接受的视觉连续性，同时避免把“平滑后的展示数据”误当作真实判定依据。

4.5 录制、回放与存档：统一的数据表达

项目在 `recording`、`replay` 与 `save` 模块中实现了录制、回放与单机存档功能。为了降低维护成本，我把三者抽象为同一条数据管线：规则推进产生状态/事件，序列化为记录；记录既可以写入文件，也可以从文件读出再驱动世界。

具体而言，录制侧记录的是“逻辑步上的关键输入/关键事件/状态快照”；回放侧则以同样的逻辑步节奏重放这些数据，从而做到“同一份数据既可用于复盘，也可用于调试”。在服务端加入单机模式后，存档可以视为“长周期的快照持久化”：当玩家退出或到达检查点时，将必要状态写入存档文件；再次进入时恢复世界并继续按同样的更新管线运行。这样设计的好处是：单机与联机并非两套完全不同的实现，而是共享绝大多数核心逻辑，只在数据来源（网络/文件/实时输入）上有所区别。

4.6 输入输出与可移植性

输入与 I/O 是“看似琐碎但最影响体验”的问题。输入方面，我在 `input` 模块中将“采集设备状态（键盘/鼠标）”与“游戏动作映射（移动/攻击/技能）”分离，使得同一套规则可以在不同输入方案下复用；同时也便于在联机模式中把动作抽象成可传输的输入消息。

输出方面，图形渲染、录制/回放文件、存档文件都属于不同形态的 I/O。为避免 I/O 直接阻塞主循环，工程上尽量将其放在独立模块中，并通过缓冲与批处理降低频繁小写入的开销。最终版本删除了专用性能测试代码后，仓库的关注点更集中在核心玩法与可交付特性上；但 I/O 分层、数据结构统一与“主循环不被杂务打断”的原则仍保留在架构中，使引擎在不同机器与不同运行模式下保持较稳定的表现。

5 工程方法与软件工程实践

5.1 Maven 与依赖管理

项目最终采用 Maven 作为统一构建入口，并配合仓库中的 Maven Wrapper (`.mvn/wrapper`) 降低环境差异带来的构建失败概率。这样做的直接收益是“可复现”：在一台新机器上，只要具备基础的 JDK 环境，就可以通过标准命令完成依赖拉取、编译打包与测试执行，而不必依赖手工拷贝 jar 或平台相关脚本。

在依赖治理上，我尽量把第三方依赖限制在“确有必要”的边界内：渲染后端需要图形库支持（项目结构中也保留了 `lib/lwjgl` 相关内容），而核心规则、场景与对象系统则尽量保持轻依赖。其好处是：核心逻辑更容易编写单元测试，也更适合在 CI 环境下稳定运行；同时当需要替换渲染后端或调整网络实现时，依赖变化被限制在局部模块，降低维护成本。

5.2 持续集成与质量门禁

仓库包含 `.github/workflows`，我将其用于最基本的持续集成：在提交或合并时自动执行构建与测试，从而形成“质量门禁”。对个人项目而言，CI 的价值不在于流程形式，而在于把错误暴露时间从“运行时发现”提前到“提交时发现”，显著降低回归成本。

在 CI 的组织上，我倾向于将步骤拆分为：拉取依赖、编译、运行单元测试、产出构建产物等。即使没有额外引入复杂的质量平台，这种最小闭环也足以覆盖常见问题（依赖缺失、编译失败、测试回归），并且能为后续扩展（例如静态检查、格式检查）留下接口。

5.3 JUnit 单元测试与覆盖率策略

项目在 `src/test/java` 下保留了较完整的测试结构，并包含 `testsupport` 用于复用测试夹具与构造场景。测试策略上，我将重点放在“最容易出错、且最值得自动化验证”的模块：数学与几何计算（例如向量/边界判定）、输入映射的规则、网络消息的编解码与边界条件、录制/存档的读写一致性等。

关于覆盖率，我的态度是“覆盖率是结果，不是目标”。对于渲染与平台强相关的代码，追求高覆盖率往往投入大、收益小；而对规则层与数据层，覆盖率与可靠性高度相关，适合优先投入。工程上我更关注的是：关键逻辑是否具备可自动回归的断言、是否能在重构后快速验证行为不变，而不是单纯追求一个数字指标。

5.4 面向对象与设计模式的工程收益

从工程角度看，本项目最大的复杂度并不来自某一个算法，而来自“对象、系统、场景、网络、回放、存档”多条功能线的长期叠加。面向对象的收益在于：可以把变化隔离在清晰的抽象边界内。

例如，`objects` 与 `components` 的拆分，使得“游戏对象是什么”和“对象拥有哪些能力”不再写死在同一个巨型类里；`scene` 负责对象生命周期与场景切换，使主循环不必理解每一种对象的细节；`net`、`recording`、`replay`、`save` 作为相对独立的子系统，让“数据从哪里来/到哪里去”成为可替换策略，而不是把网络与文件 I/O 散落在规则代码中。

在实现过程中我也有意识地使用了一些常见设计思想：用接口隔离降低模块耦合、用工厂/构造器集中管理对象创建、用事件/消息把“发生了什么”与“如何响应”解耦。这些做法并非为了堆砌模式名，而是为了让项目在多次作业迭代后仍能保持可读、可测与可扩展。

5.5 开发流程与代码质量控制

为了提高开发效率与代码质量，我采取了较为标准的开发流程控制：

- **小步提交**：尽量让每次提交都能通过构建与测试，保持主分支可运行；
- **重构优先**：当某个模块出现职责膨胀时，优先通过提取接口/拆分类/降低依赖来控制复杂度，而不是继续堆叠功能；
- **可观测性**：对网络、回放、存档等链路提供必要的日志与错误提示，使得问题可以被定位而不是“只能凭感觉调参/猜测”；
- **示例驱动**：通过 `example` 与可运行场景固化使用方式，保证引擎不是一次性代码，而是可重复启动、可验证行为的工程产物。

需要特别说明的是：在项目演进早期，我确实做过一些用于发现瓶颈的实验性代码与测试入口；但在最终提交版本中，为了让仓库更聚焦于交付的核心功能（联机、单机存档、回放/录制、完整玩法闭环），我删除了专用性能测试代码，将“可维护性与可交付性”放在首位。对应的工程收益是：代码仓库结构更清晰、入口更少、依赖更收敛，后续即便继续迭代，也能在已有的模块边界内平滑扩展，而不会引入难以维护的临时代码路径。

5.6 开发流程与代码质量控制

为了提高开发效率与代码质量，我采取了较为标准的工程流程：

- **小步提交**：尽量让每次提交都能通过构建与测试，保持主分支可运行；
- **重构优先**：当某个模块出现“职责膨胀”迹象时，优先通过拆分接口与提取类来控制复杂度，而不是继续堆叠功能；

- **可观测性**: 关键模块(网络、回放、存档)均提供可读日志与必要的统计信息,便于定位性能与正确性问题;
- **文档与示例**: 通过 README 与示例场景沉淀使用方式,使得项目更像一个可复用的引擎而非一次性作业。

值得强调的是,性能测试在早期阶段起到了“发现问题”的作用,但在最终提交版本中我选择删除专用性能测试代码,使仓库更聚焦于核心功能与可维护性。性能优化经验则保留在架构与实现中,例如系统批处理、线程池并行化与碰撞预过滤等策略,使得最终版本在不引入额外测试负担的情况下仍保持较好的运行效率。

6 课程感言与建议

6.1 收获:从语言特性到工程能力

本课程以项目驱动学习,相比于只做若干独立小练习,分阶段演进的游戏项目迫使我不断回头审视早期设计的合理性:一开始觉得方便的写法,往往会在加入并发、网络或回放后暴露出维护成本;而一次结构清晰的抽象,往往能在后续多个作业中持续受益。通过这一过程,我对 Java 的理解从“语法与 API 的掌握”逐步转向“工程取舍与边界控制”,例如何时应该引入抽象、何时应该保持简单。

课程内容覆盖面广,且与项目需求形成了较强的呼应关系:并发编程对应性能与异步写入, I/O 对应录制与存档,网络通信对应联机同步,测试与自动构建对应工程交付,设计模式则帮助稳定结构并降低耦合。这种“学完立即用、用完再反思”的节奏让我在学习上更有目标感,也更容易将知识内化为工程习惯。

6.2 建议

- 课程给出框架后,希望能安排更详细的针对参考框架的架构拆解;
- 网络联机是最难的部分之一,希望将协议设计、同步策略、插值/外推、容错与调试方法拆成更细的训练点,提供可运行的最小样例与抓包/日志指导。