轻量级 2D 游戏引擎原理及实现

高二 八班 张寅森

目录

概要

通过自己实现一个轻量级游戏引擎来探究游戏开发中的学问。

前言

说到游戏，任何人都不会陌生。就算一个人没玩过游戏，也一定听说过它。无比有趣的虚拟世界是它，令无数青少年沉沦的也是它。

但是说到游戏引擎，很多人就不知道了。

根据我的理解，这个名词指的是一类方便游戏开发的工具，它们都是面向游戏开发者的存在，与玩家无关，一般以类库的形式存在， 因此肯定也没有游戏本身知名。而像 Unity、Unreal 或者 Cry Engine，寒霜等所谓的商业引擎实际上已经超过了引擎的范畴，它们是一个完整的解决方案。

作为一种研究，我自己也实现了一个轻量级的游戏引擎。这个游戏引擎最初的版本是运行在 JVM 上的。

为什么是 JVM（Java Virtual Machine，Java 虚拟机）呢？首先我是考虑到跨平台性，另外我最熟悉的平台就是 JVM 了。由于著名沙盒游戏 Minecraft 在 Mojang 时代的版本就是运行在 JVM 上的，因此该游戏的玩家的电脑上都装有 JRE。很多喜欢玩 Minecraft 的用户就可以直接在自己的电脑上运行可执行的 jar 文件。还有就是 JVM 字节码可以很方便地编译到微软的.Net 平台字节码，因此要想脱离 JVM 运行也很方便。其实更多地是考虑到我自身最熟悉的平台就是 JVM，因此遇到一些非常规问题（比如编译器的 bug，或者虚拟机架构）我依旧可以解决。

但是 JVM 有一个缺点，就是它不能手动 GC（Garbage Collection，垃圾回收）。JVM 的 GC 是由虚拟机自动完成的，并且在 GC 过程中会有短暂地卡顿，因此会造成一定程度上的不稳定。这并不是一个大问题，在原理实现相同的情况下可以使用提供手动 GC 的平台或语言重新实现一个即可。

因此，在后来，我又使用 C#在 CLR（.Net 平台）上实现了另一个版本。API 除了命名为了适应微软的官方规范之外（比如接口 I 开头，方法名使用 Pascal 命名法而不是驼峰）和 JVM 版本都是一样的。因此使用过 JVM 版引擎的人可以很快适应 CLR 版。

为什么是 2D 呢？虽然 Java 本身是有 3D 库的，但是事实上，JRE 并没有包含 Java3D 的运行时，它是单独打包的。开发包也是外部提供。这就为部署和支持增加了难度，因此我暂时地放弃了 3D。

语言的选择并没有让我犹豫那么多。我是肯定不会选择原生的 Java 语言的，一是因为 Java 这门语言本身名声不是很好，二来一直被诟病的 Checked Exception 和糟糕的 Functional Programming 支持让我对这门语言抱有一定程度的偏见。好不容易在第八个版本的实现中有了一个 Lambda，却只是一个匿名内部类的语法糖罢了。虽然这个语法糖的引入让许多爱好 FP 的 JVM 开发者重见阳光，却依旧不能和方言中的 FP 媲美。而许多 JVM 方言中，我最偏爱的就是 Kotlin 了。非常轻量的运行时（不算反射库的话只有 732kb，这点优于任何别的语言），优秀的 FP 支持（更多是思想上的，这点优于 Java），与 Java 更无缝的对接（这点优于非 OOP 的 Clojure），以及现代语言就算是为了面子也必须有的编译期静态空指针保护（这点优于 Scala）。而且这门出现于 2013 年的语言还有很大发展空间，不像 Xtend 那样不知死活。因此我选择了 Kotlin。

以上是与引擎本身无关的内容。下面说说引擎本身。本文所有与引擎有关的信息可能与现有的主流引擎实现方式不同，因为我并没有读过任何关于游戏引擎实现的书，而更多地是

依靠自己道听途说的一些技巧，剩下的自己动脑思考解决办法。整个引擎所有代码未经特别标注处全部由个人编写，因此本文几乎是没有参考资料的，也不存在"指导老师"一说。

整体结构

一个游戏引擎一定要有一个严格的架构。我个人之力也不可能像商业引擎那样制作完善的设计器（事实上，那些设计器的源码远多于引擎本身）受到之前学 Android 原生开发的影响，我也采用了类似生命周期的设计。设计时，我首先站在开发者的角度考虑问题，再设计游戏主类。声明：

```
open class Game constructor()
```

首先游戏主类继承一个含有生命周期的类 ,就像 Android 的 Activity 一样——然后通过重载生命周期方法来在特定的时刻执行特定的代码。首先我设计了初始化、刷新时、关闭时三个最主要的生命周期方法，让开发者可以初始化变量和元素以及窗口信息，可以在每次刷新时更新数据，以及在窗口关闭时进行一些操作，比如保存数据、确认关闭等。然后是窗口失去焦点、获得焦点、窗口被鼠标点击以及其他鼠标事件，加上之前的三个，一共七个生命周期方法。因为开发者不一定重载所有生命周期方法，如果将这些方法声明为抽象、全部强制重载的话，只会导致代码量无谓增加，就像 Checked Exception 一样。因此我没有把它们声明成抽象类，而是方法体为空的方法。方法签名如下：

```
open fun onExit() = Unit
open fun onInit() = Unit
open fun onRefresh() = Unit
open fun onClick(e: OnClickEvent?) = Unit
open fun onFocus(e: OnWindowEvent?) = Unit
open fun onLoseFocus(e: OnWindowEvent?) = Unit
open fun onMouse(e: OnMouseEvent?) = Unit
```

然后是游戏主循环。任何引擎在运行过程中都是有一个不停刷新的线程，这个线程被称为主线程。它需要满足以下几个条件：

非暂停时期，不停地刷新数据和画面。

尽可能早地开始进行刷新。

捕获一切异常，防止因不必要的原因退出游戏。

所以，首先让游戏主类实现 Runnable 接口并实现 run 方法，在里面实现主循环的数据刷新（onRefresh）以及画面的重绘。为了方便以后处理暂停，这里引入一个 paused 的布

尔变量，表示暂停。

```
open class Game() : Runnable {
    override fun run() {
        while (true) if (paused) {
            onRefresh()
            panel.repaint()
        }
    }
}
```

然后在主构造方法的初始化代码块中创建线程。

```
init {
    Thread(this).start()
}
```

同样，可以通过这种匿名接口的写法来实现上面两个代码框中实现的功能，代码量减少了三行，相对美观一些：

```
init {
    Thread({
        while (true) if (paused) {
            onRefresh()
            panel.repaint()
        }
    }).start()
```

这样，就能实现主线程的刷新了。但是这样的刷新是不稳定的刷新，因为你不能控制刷新的频率。如果频率不受控，过快的频率可能会导致大量的临时变量产生，出现过多的内存垃圾触发了 GC 就会导致卡顿，降低用户体验。因此首先要控制刷新频率。那么这里就涉及游戏的计时器。

计时器

最初我直接使用了 Thread 类的 sleep 方法来使主线程暂停。

```
Thread.sleep(10)
```

这种方法是绝对不行的。sleep 方法的原理是阻塞线程，可能会导致内存垃圾、消耗 CPU 等后果，而且阻塞线程的话，窗口中鼠标会一直处于"加载中"的状态，无法响应鼠标事件（这点仅在 CLR 版上有所体现，有可能是因为 CLR 和 JVM 的线程模型不同）真正的游戏引擎应该使用计时器（Timer）。在我的引擎中的实现实现大概是这样：

```
open class Timer(protected val time: Int, times: Int) {
    constructor(time: Int) : this(time, -1)

    var times = times
        private set
    private var start = System.currentTimeMillis()

    fun ended(): Boolean = if (System.currentTimeMillis() - start > time && times != 0) {
        start = System.currentTimeMillis()
        if (times > 0) times--
        true
    } else false
}
```

这种不含任何线程操作的计时器不会有任何性能上的问题。它不会阻塞线程，而且它不像 sleep 方法那样——强制停止指定毫秒数，而是真正意义上的计时。如果引擎在执行延时操作，它不会干扰计时。

举个例子。现在引擎正在执行一个延时操作，需要 1ms。然后你需要进行计时，每 3ms 执行一次该操作。如果你使用 sleep(3)，那么真实情况是，线程被阻塞了 4ms。但是如果你使用计时器 Timer(3).ended()，那么这耽误的 1ms 并不会造成任何影响。线程依旧只会停止 3ms。并且不是阻塞。

出于方便考虑，上述代码实现的计时器有两个构造器，其中一个还为计时器增加了计数功能——执行指定次数后就不再执行计数了。这样如果开发者只需要执行一定次数，就不需要再手动判断了。

在引擎中使用计时器，只需要在初始化的时候构造，然后每次刷新的时候调用 ended 方法，如果返回 true，那么就执行那段定时执行的方法吧。

最后，在引擎的主线程的刷新过程中添加一个 2ms 的计时器，只要硬件性能满足，就能实现 500fps 的刷新了。事实证明，我的引擎就算是 2000fps 也不会有卡顿。正常情况下的 2D 游戏引擎都是 1000fps 左右的，尤其是 C++游戏引擎。就算是 3D 引擎，普通的也有 100fps 左右。但是经过我多次的测试，刷新时间间隔太短（500~1000fps 左右的时候）会导致与时间有关的计算精度产生偏差（比如加速度和碰撞）（据个人推测，可能是虚拟机的内部问题，不过这种极高精度上的误差本身也不是特别严重，因此 JRE 并没有将它的精度优化到极致，对于开发者来说也是可以接受的），最终我把引擎锁定在了 250fps，也就是 4ms 一次刷新。这样刷新就没有任何问题了。

这同时也证明了 JVM 的执行效率。在做这次 fps 的测试之前，我一直以为 JVM 的效率很

低。事实上，这种间接调用系统 API 的行为并没有过多的降低 JVM 的效率。一个能流畅地进行 1000fps 的刷新的 VM，效率上是没有什么可诟病的地方的。唯一的缺点，可能就是无法手动管理内存导致偶尔 VM 进行大量 GC 导致的卡顿了（目前未发现因此造成的大型性能问题）。

游戏对象 CRUD

主循环实现了，接下来需要实现游戏对象的 CRUD（添加、删除、更新、查询，简称增删改查或 CRUD）了。游戏对象考虑如下设计：

首先写一个基类，或者接口，它含有游戏对象所必须的方法，和属性。方法比如判断是否包含一个点，是否与另一对象产生碰撞，绘制方法等；属性比如横纵坐标、长宽、是否已经死亡（便于回收）等。在引擎主类中有一个列表（List），这个列表还必须是链表实现的（LinkedList），方便从列表中间进行频繁的插入和删除。数组实现（ArrayList）相应的操作会显得相对较慢。

然后在引擎主循环中分别调用碰撞检测的方法、根据各自的动画进行移动、缩放、旋转，然后分别进行双缓冲的绘制。关于双缓冲后文会讲到。

这样的设计整体看起来是没有问题的。但是当引擎在支持一个复杂的游戏时，就会出现很多看起来并不致命的 Exception——大概是由于在遍历一个列表时对它进行插入与删除操作所造成的。

下面是我测试引擎时一个 Demo 所出现的 Exception：

```
Exception in thread "AWT-EventQueue-0":
        java.util.ConcurrentModificationException
            at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
            at java.util.ArrayList$Itr.next(Unknown Source)
            at org.frice.game.Game$GamePanel.paintComponent(Game.kt:222)
```

这样的情况让我很困扰，因为我并没有进行任何可能导致程序崩溃的操作（比如访问空指针），引擎没有什么问题，Demo 也没有什么问题（事实证明，这一系列的 Excepttion 不是由任何逻辑上的问题产生的）。最后我才发现，是由于在不当的时候处理对象的增删导致的。正确的实现应该是，为增和删分别准备一个缓冲用的 List，每次增的时候就往增的缓冲 List 里面添加元素，删的时候往删的缓冲 List 里面添加元素。最后在完成所有对象的碰撞

检测、动画处理以及绘制之前执行对应的列表删除和增加的操作。这样，任何时候进行增删都不会直接涉及对象列表的增删，而是只操作缓冲 List，保证不会在遍历时产生错误。代码实现如下：

```
protected @JvmField val objects = LinkedList<AbstractObject>()
protected @JvmField val objectDeleteBuffer = ArrayList<AbstractObject>()
protected @JvmField val objectAddBuffer = ArrayList<AbstractObject>()

protected @JvmField val timeListeners = LinkedList<FTimeListener>()
protected @JvmField val timeListenerDeleteBuffer = ArrayList<FTimeListener>()
protected @JvmField val timeListenerAddBuffer = ArrayList<FTimeListener>()

protected @JvmField val texts = LinkedList<FText>()
protected @JvmField val textDeleteBuffer = ArrayList<FText>()
protected @JvmField val textAddBuffer = ArrayList<FText>()
```

以上是所有对象的声明以及初始化。然后在每次刷新时，执行以下代码以处理所有的缓冲、对象列表：

```
objects.addAll(objectAddBuffer)
objects.removeAll(objectDeleteBuffer)
objectDeleteBuffer.clear()
objectAddBuffer.clear()

timeListeners.addAll(timeListenerAddBuffer)
timeListeners.removeAll(timeListenerDeleteBuffer)
timeListenerDeleteBuffer.clear()
timeListenerAddBuffer.clear()

texts.addAll(textAddBuffer)
texts.removeAll(textDeleteBuffer)
textDeleteBuffer.clear()
textAddBuffer.clear()
```

以上代码先分别处理所有的缓冲，然后再清除所有缓冲。这样就不会出现上文所述的 Exception 了，并且可以保证安全的 CRUD。

双缓冲（DoubleBuffer）

DoubleBuffer 可谓是任何涉及 GUI 的框架所必须的技术，它可以保证 GUI 画面的流畅显示。同时，利用 DoubleBuffer 可以很方便的实现截图的保存与获取。由于这个游戏引擎是使用了 Swing 的窗口类的，因此只需要重载 paintComponent 方法就能使用 Swing 自带的 DoubleBuffer。Swing 的双缓冲是在 paint 方法中实现的，因此如果重载 paint 方法，自

带的双缓冲方法就消失了。所以这里推荐重载 paintComponent 方法，自带的 DoubleBuffer 就不会消失了。不过如果只使用自带的 DoubleBuffer 的话就不能获取截图了，因此我还是重新实现了一次。

在上文的基础上写出来的简单引擎在较高频率刷新的情况下是会有很严重的"闪屏"现象的——就算是刷新不是很快的时候（100fps 左右），屏幕也会经常间歇性地在一瞬间变成白色，然后又恢复正常。

这种现象出现的原因是可以很容易地解释的。引擎有很多部分可能造成这种现象，但最主要的罪魁祸首是下面这行代码：

```
g.clearRect(width, height);
```

因为每次重绘的时候，为了避免上次绘制的对象依旧保留在屏幕上，首先必须清空整个屏幕，然后再分别绘制每个游戏对象。这看起来没什么问题，但是绘制对象是需要时间的。当引擎清空屏幕后、每个游戏对象又没有全部完成绘制时，如果这一时间稍长，就会产生上述的"闪屏"现象。

"闪屏"这种现象是绝对不允许的——它不仅仅使流畅度下降，甚至会在生理意义上伤害用户的视觉系统。解决"闪屏"的方法就是实现游戏画面的 DoubleBuffer。

何为"DoubleBuffer"？顾名思义，就是 Doubled Buffer（二次缓冲）。将要绘制的游戏画面先在内存中绘制出来，然后再一次性将新的画面应用到屏幕上。这样，每次涉及屏幕的绘制时，根本就不会出现上文清屏的操作——它已经在内存里进行过清屏了。并且每次向屏幕绘制时，所有对象都已经绘制完毕。这样能大大提高引擎的绘制效果。接下来是代码实现。

首先，游戏类新增一个 buffer 属性，类型是 BufferImage，一个可以进行像素级绘制的 Image 类的子类。它提供了 setRGB(x: Int, y: Int, rgb: Int)以及 getRGB(x: Int, y: Int)等方法来操作每一个像素的 rgb 值，还可以获取表示图片的 byte 数组。它可以通过 ImageIO 类从图片中获取，也可以通过长、宽和 Image 类型来直接构造一张空的图片。这里的缓冲图片通过长宽构造。

```
private val buffer = BufferedImage(
        panel.width, panel.height,
        BufferedImage.TYPE_INT_ARGB)

private val bg: Graphics2D
        get() = buffer.graphics as Graphics2D
```

.然后在每次刷新的时候，获取 bg 对象（即 buffer 的 graphics 对象，提供了很多绘制

相关的 API），最后进行每一个对象的动画处理、碰撞检测、绘制即可。绘制完成之后，将 buffer 绘制到控件中。准确来说，为了保证截图的稳定性，这里应该再添加一个 stableBuffer 对象，每次 buffer 绘制完之后在 stableBuffer 中保存一次。这样提供截图 API 就很方便了——调用截图 API 时，复制一份 stableBuffer 返回即可。

```
override fun update(g: Graphics?) = paint(g!!)
override fun paintComponent(g: Graphics) {
    drawBackground(back, bg)
    drawEverything({ bg })

    stableBuffer.graphics.drawImage(buffer, 0, 0, this)
    g.drawImage(buffer, 0, 0, this)
}
```

重新运行。可以看到，“闪屏”现象消失了。这说明了，通过 DoubleBuffer 能同时保证流畅的刷新和较稳定的截图了。DoubleBuffer 在 GUI 领域是非常普遍的技术，实现简单，效果拔群。Swing 有内置的 DoubleBuffer，.Net 的 Win Form 类还提供了 DoubleBuffer 相关的 API。

到现在，这个引擎的绘制部分已经成型了。

对象模型与资源管理

一个游戏是由很多部分组成的——我和 Unity 一样，称之为游戏对象。

不过在说游戏对象之前，这里先引入一个概念——“Sprite（精灵）”，它们可以看作是一种资源，比如图片。而如果你需要将这种资源应用到游戏中，就需要使用游戏对象来呈现它。比如，在 Unity 里面，这种具有行为的游戏对象叫做 gameObject。在一个 Unity 的 MonoBehavior 里面可以直接获取当前 gameObject。比如（代码是 C#）：

```
public class Move : MonoBehaviour
{
    public void Start()
    {
        Destroy(gameObject);
    }
}
```

对象模型对一个游戏引擎来说是十分重要的。它决定了所有基于该引擎的游戏组织代码的方式。代码的组织是非常重要的，一团糟的代码组织会大幅降低开发者生产力。而 API 的设计也同样会大幅影响开发效率——一个典型的反面教材就是 JRE 提供的 Java 标准库。臃肿的 API 设计让追求代码美感的程序员对 Java 嗤之以鼻。Checked Exception 所造成的代码噪声也同样是一个糟糕的设计。而这方面的典范就是许多有能力构造 DSL（Domain Specific Language，领域特定语言）的语言了，比如 Ruby（Rails），Groovy/Kotlin（Gradle）等语言/平台通过 DSL 式的语法为开发者提供了相当的便捷。因此，API 的设计是越简单越好的。

在我看来，出现在游戏画面上的对象和那些对象所呈现出的资源是应该分开的。就像上文说的 gameObject 和 Sprite 一样。我将游戏画面上的对象称为 Object，并编写对应基类 FObject，然后又为了方便编写按钮、文字等较为特殊的控件，我又封装了 AbstractObject 和 PhysicalObject。然后还有对应的资源类，基类是 FResource，子类有 ImageResource、ColorResource 等。音频播放被我忽略了，并另外单独封装到 AudioPlayer 类中。

为什么要将游戏对象和资源分开呢？除了仅仅是设计上的考虑，我个人认为是因为这样可以更好的管理内存。假如现在有十个对象，每个对象持有一个相同的图片引用。如果我将资源和对象绑定起来，那么我每次创建一次对象都需要加载一次资源。过一段时间就会产生大量的内存垃圾——因为重复加载了大量的图片，而且这些图片是相同的。这样明显是不好的。虽然游戏开发者也可以通过手动让游戏对象持有同一个图片的引用来节约这段内存，不过如果图片是通过文件路径构造的话就做不到了。再考虑加密资源的引用，开发者如果想手动管理内存，将付出巨大的时间和精力。

而且，就设计上来讲，一个好的引擎绝对不能让开发者手动处理任何广义的“琐事”。那么，把一切资源交给游戏引擎来管理是一种不二的选择。

考虑到多态的设计，首先编写资源的基类——FResource，和对象基类 FObject。考虑到部分对象不存在长宽的概念，单独封装 AbstractObject，并使 FObject 实现 AbstractObject 接口。然后针对基于几何图形的 gameObject 和基于图片的 gameObject，分别封装 ShapeObject 和 ImageObject 类，提供相应的构造方法、属性，以及静态的工厂方法。