**中级软件设计实作**

**简单的游戏引擎 技术文档**

**xxxxxxxxxxxxxx xxx**

**目录**

1. **题目……………………………………………3**
2. **设计思路………………………………………4**
3. **细节探讨………………………………………6**
4. **技术讨论………………………………………9**
5. **运行截图……………………………………10**

**（1）题目:** **简单的游戏引擎**

题目需求：

游戏引擎中，读取一个游戏定义文件

该游戏定义文件中包括：定义场景地图，定义角色，角色的起始位置，定义事件（当角色移动到什么位置时或有什么动作时，执行一个什么操作），定义场景物品

游戏引擎读取这个游戏定义文件，启动游戏运行。

用户可在游戏引擎的支撑下，玩这个游戏

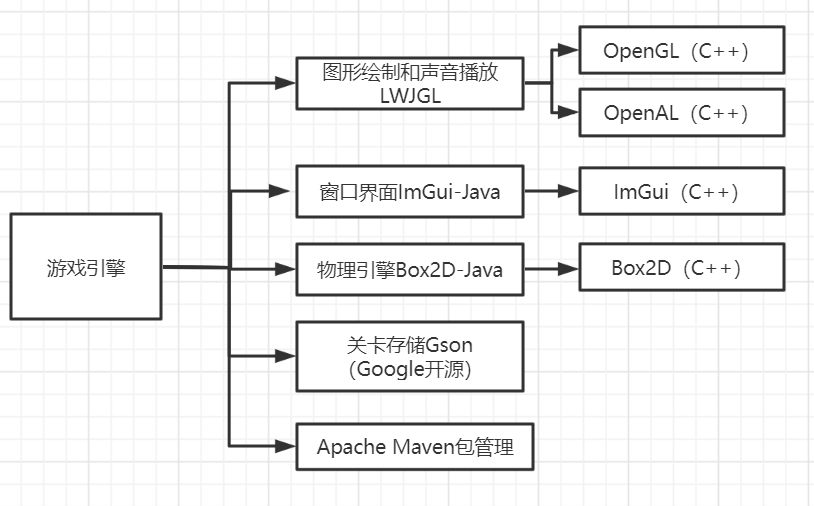
技术准备：

游戏引擎的功能

代码加载图片的方法

**（2）设计思路**

1、本游戏引擎的项目结构图如下：



开发语言：游戏引擎使用Java语言开发，编译器为JDK17，Java语言标准为JDK17。

图形绘制：对于图形绘制，这里选择OpenGL作为图形绘制库，但是因为Java没有办法直接调用基于C++的OpenGL的API，所以这里使用了LWJGL这一个转接库，全称为Light Weighted Java Open Graphics Library，它是一个轻量级的包装库，使用JNI将原来基于C++的OpenGL接口转接到Java中使用，方法名和函数参数列表都没有改变，这样就可以原封不动的访问OpenGL库中的接口了。

声音播放：对于声音播放，这里选择OpenAL作为声音播放库，同理，由于Java无法直接调用基于C++的OpenAL的API，所以同样使用了LWJGL库提供的转接方法，通过JNI原封不动的将参数转发到OpenAL的API中调用。

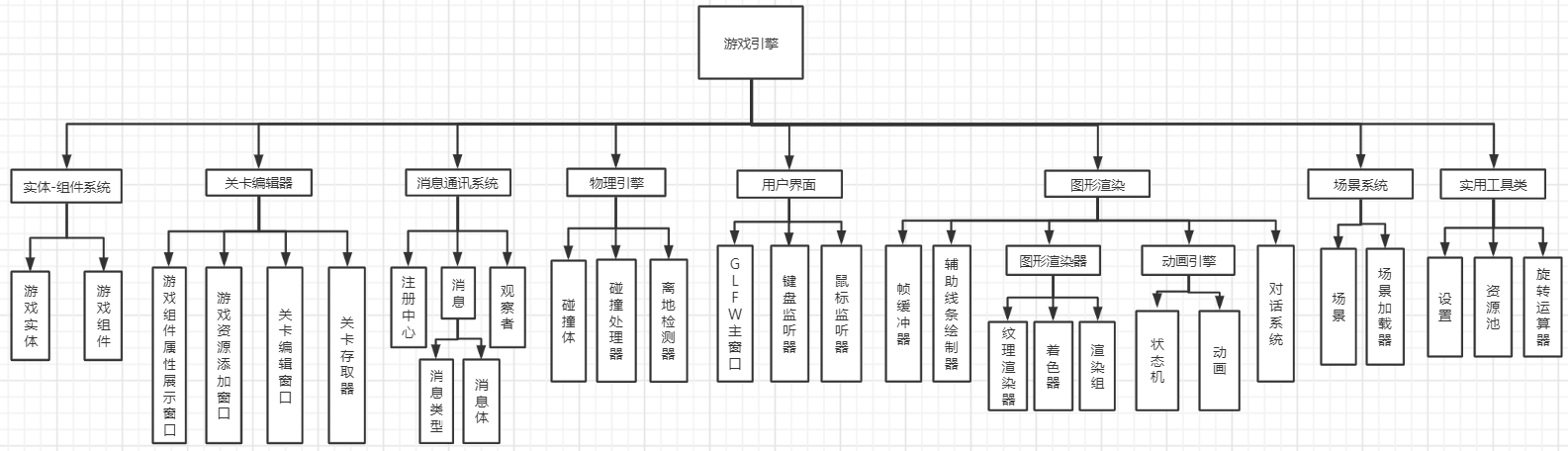
图形界面：对于图形界面的部分，这里选择ImGui作为图形库。同理，该图形库是基于C++写的，无法被Java调用，所以需要使用基于JNI的转接库ImGui-Java。

物理引擎：对于物理引擎的部分，这里选择Box2D作为物理引擎库。同理，需要使用转接库Box2D-Java。虽然使用了第三方库Box2D，但是游戏引擎中只让Box2D去作检测碰撞的用途，检测到碰撞后的具体处理逻辑是由用户编写的。关于物体移动的速度和下降的速度与加速度等是由游戏引擎自身来控制的，在速度上Box2D时刻同步于游戏引擎，这部分的控制权不在Box2D中，控制权在于用户手上。

关卡存储：对于关卡的存储使用谷歌的开源项目Gson，它可以方便的将游戏中的物品序列化为json格式并存储到文件中。将来读取文件时也可以反序列化后恢复游戏中存储的场景·。

包管理：对于包管理工具，使用的是Apache基金会开源的Maven工具，可以很方便的将所需的要的包从Maven中央仓库下载到本地使用。同时编译打包的工作也是由该包管理工具支持。最终项目生成的是通过Maven工具打包的Jar格式文件。

2、本游戏引擎的组件结构图如下：



实体-组件系统：

游戏引擎采用了实体-组件系统，场景中的每一个游戏物品都是一个实体对象，而每一个实体对象都由许多的组件组成。可以随时从某一个游戏物品中根据Java反射机制获取这个游戏物品的某个组件，这样就让游戏对象的扩展性非常高，用户可以将自己喜欢的对象定义为组件的子类，然后将其加入到游戏对象中。

关卡编辑器：

关卡编辑器的部分，游戏组件属性展示窗口主要用于提供游戏属性的修改功能，例如修改游戏物品的位置，伸缩大小，旋转角度，层数等等。

游戏资源添加窗口主要利用于根据用户自定义的资源以及用户的需求将游戏物品添加到场景中，例如实体方块，装饰方块，游戏玩家，游戏怪物以及游戏道具等。

关卡编辑窗口主要用于选取场景中的游戏物品并利用Java反射机制将这个游戏物品的属性暴露到游戏组件属性展示窗口来提供给用户修改。并且该关卡编辑窗口还可以框选多个游戏物品，同时移动多个游戏物品，改变多个游戏物品的层数等。

关卡存取器主要用于将当前编辑好的关卡保存到文件中，不仅如此，用户还可以利用关卡存取器将文件中的关卡读取出来继续编辑。

消息通讯系统：

关于消息通讯系统，使用的是观察者模式，首先需要接收消息的对象先到注册中心注册自己，成为观察者。每当有消息到来时，每个观察者都会被通知到，确实需要接收消息的再接收，不需要接收的就直接忽略。消息有消息类型和消息体，消息体是用户自定义的内容。用户可以指定自己的消息通讯内容。引擎主要使用消息通讯系统来进行游戏的开始与结束，关卡的读取与保存等操作。

物理引擎：

关于物理引擎，由于使用了Box2D，所以要在游戏引擎中创建一些与物理世界所对应的对象。比如碰撞体，游戏引擎中定义的碰撞体有圆形碰撞体，长方形碰撞体以及上方下圆的碉堡形碰撞体。

碰撞处理器主要是在物理引擎检测到物体发生碰撞时的回调函数，游戏引擎的处理是将借助这个碰撞回调将工作转发给对应的游戏对象的碰撞函数，这些碰撞函数由用户去自定义。

离地检测器主要利用了碰撞处理器去判断某个游戏物品在不在地面上，这个机制非常的重要，比如用户可以根据游戏对象在不在地面上而切换人物姿态。

用户界面：

用户界面主要利用ImGui来实现一些用户界面的功能。

游戏的整一个窗口的界面是基于OpenGL库的glfw实现的，使用glfw创建窗口并将窗口挂接给ImGui管理，这样就成功将OpenGL接入了ImGui。

在将OpenGL接入ImGui后，安装对应的鼠标按键和移动的回调函数，实现对鼠标移动和按键的监控，这样就可以实时获取鼠标的位置和按键，从而决定在窗口内执行什么功能。比如在关卡编辑器中，按住鼠标中键并拖动游戏预览窗口就可以移动摄像机的位置。

对于键盘的回调同理，安装对应的回调函数，实现对键盘的监控。这样就可以实时获取用户按键的情况，从而决定在窗口内执行什么功能。比如在关卡编辑器中，选中游戏对象后按下del键可以将该游戏对象从场景中移除。

图形渲染：

帧缓冲是一个非常重要的技术，利用帧缓冲器，可以很方便的先将画面离屏渲染到帧缓存中，然后再将帧缓存中的内容读取到屏幕上作为纹理图片输出，这样就可以以相同的分辨率渲染，然后再伸缩到不同的屏幕大小上，实现游戏画面大小的动态调节。另外一个使用到帧缓冲的地方是游戏物品的选取，借鉴了3D游戏引擎的物品选取技术，使用两次渲染技术，第一次做颜色渲染，然后提取深度，制作成选取材质，用户点击鼠标时从该材质中提取对应的像素点，进行颜色深度分析，映射到对应的游戏物品上。第二次渲染到真实的帧缓存中，显示到屏幕上。

辅助线条绘制器的用途是在关卡编辑界面提供基本的网格线条绘制以及物品碰撞体的绘制，方便用户更改物品属性时能及时的反馈到屏幕上，这种视觉反馈能够很好的辅助用户进行精确的关卡设计。

图形渲染器由三个部分组成，分别是纹理渲染器，着色器，渲染组。纹理渲染器主要功能是负责在GPU中开辟内存，并将用户的纹理渲染到屏幕上。着色器主要是负责编译和运行GPU脚本，给用户的纹理上色，分为顶点着色器和片段着色器。渲染组是将相同深度的多个纹理、材质放在一个渲染组中，一次GPU绘画函数调用使用着色器绘制多个纹理和材质，这样就可以达到批量绘制的效果，提高GPU的运行效率，提高运行时帧率。一个图形渲染器有多个渲染组，这些渲染组的渲染方法被图形渲染器逐个调用。

动画引擎由状态机和动画组成，动画主要是保存多张图片并保存每一张图片的显示时间长短。每次更新时都检查是否需要切换图片，这样就可以实现图片的播放功能，类似动画。状态机主要是保持了着各个状态之间的相互转换关系，一个状态可以经过什么条件转换到另外一个状态，展现在屏幕上就是一个动画在什么条件下转换到另外一个动画，比如离地检测检测到离地后从静止状态转变为跳跃状态。

对话系统是游戏的角色进行对话的系统，主要是保存当前对话的内容和当前进行的对话的进度，对话框默认出现在游戏对象的上方。每帧更新，默认不显示对话框，但是如果触发条件满足，对话就会开启。对话完成后对话框自动消失。

场景系统：

场景系统分为场景和场景加载器。场景是固定的，场景拥有许多游戏物品，可以实时往场景中添加物品，也可以根据名字，UID，拥有的组件类型等从场景中获取某个游戏物品。场景有从文件中加载和保存到文件中的功能。场景在加载时调用场景加载器进行资源的初始化和游戏内初始物品的初始化。

场景加载器主要是用户自定义的加载器，在这个加载器中用户可以选择加载哪些资源到资源池中，并将这些资源将怎样被加载到场景中。通过使用加载器的init方法对游戏场景的相机，初始物品等进行初始化。场景只有一个，但是通过不同的加载器可以实现多个场景之间的切换，这样就达到了场景与场景之间高度松耦合的效果。

实用工具类：

设置类是用来给用户提供设置使用的，在里面可以设置屏幕的显示分辨率，该值默认与屏幕分辨率匹配。可以设置关卡编辑器的格子的宽高，可以设置鼠标移动灵敏度，也可以设置鼠标滚轮灵敏度。该类有一个静态代码块，在一开始就初始化glfw窗口并获取屏幕分辨率。

资源池是一种很好的池子，可以将已经加载好的资源反复利用，而不用反复的加载资源。比如着色器，基本上每一个渲染组都要有一个着色器，如果每新建一个渲染组都构造一个着色器对象的话，需要反复编译和运行多个着色器脚本，但是这种工作是没有必要的，每个着色器脚本只需要编译一次就可以多次使用了。包括声音和图片等资源也同理，不需要多次在内存中开辟空间存储音频和图片，同一段音频和同一张图片只需要加载一次即可多次使用。

旋转运算器，旋转运算器是一个特殊的工具类，主要用于计算平面直角坐标系中的某一个点经过一定角度的偏移后到达了哪一个位置，主要运用了一些三角函数公式来推导计算，具体参考了：[求在平面直角坐标系中，一个点绕坐标原点旋转一定角度后点的坐标\_lvchang-CSDN博客\_点坐标旋转一定角度后的坐标](https://blog.csdn.net/sinat_32560085/article/details/106389000)中的计算方法。

以上就是游戏引擎项目的一些大概的设计思路、项目整体的项目结构以及项目的组件结构，至于这些内容的部分具体实现细节将在第（3）节的技术探讨中仔细讨论。

**（3）细节探讨**

实体-组件系统：

实体即GameObject，这个GameObject中有一个List<Component>，组件即是Component，所有游戏的组件都继承自Component，这样就可以将游戏组件加入到GameObject中。用户自定义的组件只要继承自Component都可以被成为场景的游戏对象的组件，被游戏引擎更新。

关卡编辑器：

关卡编辑器的属性窗口主要是对在当前在关卡编辑窗口中选中的游戏对象，利用Java反射机制分门别类调用ImGui的对应方法将游戏组件的属性展现在关卡编辑器中，用户可以很方便的作修改。

游戏资源添加窗口主要是利用资源池AssetPool和根据用户自定义的imgui方法将用户所需的资源加载到窗口上，用户可以像生成控件一样将这些物品添加到游戏场景中。

关卡编辑窗口主要利用了DebugDraw类绘制辅助线，并借助MouseControl类和KeyControl类来实现对关卡编辑器的鼠标和键盘功能的实现。而MouseControl类又依赖于MouseListener中的方法，KeyControl类又依赖于KeyListener中的方法。而MouseListener和KeyListener又在ImGui中注册了监听事件。

关卡存取器主要是使用Gson的fromJson和toJson方法结合Java的File类实现对游戏对象和组件的保存和载入，由于GameObject和Component比较特殊，Component由于是基类，在序列化是无法确定是哪一个子类的对象，所以需要将类名一起保存起来，而GameObject因为需要在Component中存一个引用，所以也需要自定义反序列化器。所以需要自定义GameObjectDeserializer和ComponentDeserializer这两个类，其中Component自定义了序列化和反序列化的方法，GameObjectDeserializer只是定义反序列化的方法。

消息通讯系统：

注册中心是EventSystem，其中定义了List<Observer>，里面存储着所有注册到这个消息通讯系统的游戏对象，用户可以利用addObserver方法往注册中心注册自己。想要发出消息时可以使用消息通讯系统的notify方法。第一个参数指定接收该消息的游戏对象，如果第一个参数为空默认为发个所有的游戏对象。

消息为Event，消息内包含了消息的类型和消息的本体，这样接收该消息的游戏对象可以根据消息的类型做出不同的操作，用户也可以自定义消息，自定义消息体，只需要继承Event类就可以了，该消息通讯系统具有可扩展性。

所有注册到注册中心的类都必须实现Observer接口，实现这个接口需要实现接收到消息后的处理函数。

物理引擎：

在Box2D中Fixture代表物体, PolygonShape和CircleShape代表碰撞体。所以在游戏引擎中创建了RigidBody2D作为Fixture的对应体，里面有速度，加速度，角度，旋转等量可以设置，并且这部分量是由用户控制的，使用对应的setter方法设置后就会将这些量同步设置进Box2D中，这样使得Box2D的世界中的物体同步于用户想要的物体。而PolygonShape对应于Box2DCollider，CircleShape对应于CircleCollider，这两个类主要是用于提供给用户添加碰撞体使用的，添加了碰撞体后就会在Box2D中同步添加对应的碰撞体，而碰撞检测的工作是由Box2D来做，检测到物体之间的碰撞后就交由游戏引擎在Box2D注册的ContactListener碰撞处理器来处理，这个类会将这个碰撞的信息转交给对应游戏对象的处理函数去处理，而这些处理函数由用户自定义完成。

离地检测的工作主要由Box2D的RayCast线条触地检测来完成，Box2D将这些信息回传到游戏引擎的RayCastInfo中，从RayCastInfo中提取信息就可以得知游戏物体有没有离地。

用户界面：

刚开始进入的游戏窗口是使用OpenGL的glfw的glfwCreateWindow构建的，在构建窗口的同时使用glfwSetCursorPosCallback、glfwSetMouseButtonCallback和glfwSetScrollCallback。初始化KeyListener和MouseListener监听器，使用glfwMakeContextCurrent将OpenGL与当前窗口挂钩。在此同时使用OpenAL的alcGetString获取音频设备，alcOpenDevice初始化音频设备，alcCreateContext和alcMakeContextCurrent设置将窗口与音频设备挂钩。接着初始化ImGui层，将glfw窗口与ImGuiGlfw挂钩，所有的用户图形界面绘制工作交由ImGui来完成。

完成这些初始化工作后就开始主循环，在主循环中遍历并更新所有的游戏物品。为了实现关卡编辑器的鼠标点击选取功能，需要经过两次渲染，第一次先将游戏物品UID对应的颜色渲染到PickingTexture上，需要做物品选取的是否就直接在PickingTexture上读像素点，需要框选多个物品时也是读像素点，不过需要读取一个区域的像素点然后使用Set去重。对于场景中的空白，使用-1表示，这样在获取物品时先将UID整体向右偏移1，然后只需要检查UID是否为0即可。在第二次渲染开始之前，先做游戏物品的更新，接着第二次渲染就离屏渲染到帧缓存中，然后由ImGui将帧缓存中的帧显示到GameViewWindow中，完成游戏画面的显示。这样就可以实现渲染两次，更新一次。

图形渲染：

帧缓存是OpenGL3.0添加的功能，使用glGenFramebuffers生成帧缓存，使用glGenRenderbuffers生成渲染缓存，最后再使用glFramebufferRenderbuffer将他们绑定在一起。需要渲染时使用glBindFramebuffer就可以将渲染的工作先渲染到在GPU中开辟的一块内存区域中，然后再从这块区域以纹理的形式读取出来画到屏幕上。这样做的好处是每次渲染的分辨率非常稳定，不会随着用户界面的改变而改变，其次是想要实现关卡编辑器中的物品选取功能必须要帧缓存的支持。对于物品选取，使用的着色器是picking.glsl

辅助线条绘制系统是由DebugDraw类负责的。该类中使用直线着色器debugLine2D.glsl，采取与帧缓存渲染差不多的方法glDrawArrays来画到屏幕上。对于圆形的绘制，采取的是近似法，从圆上采样20个点来绘制20条线，连接形成一个圆。

图形渲染器使用default.glsl作为着色器，利用glDrawElements以三角形采样的形式将图形绘制到屏幕上，而其中为了提高渲染效率又使用了渲染组RenderBatch，每一个渲染组中渲染的物品的数量由GPU所能支持的最大纹理数以及本渲染组中已有的渲染物品数量决定，每一个渲染组中都保存着多个纹理渲染器。渲染器Renderer在每次更新，render方法被调用时，都会调用每一个RenderBatch中的render方法进行组渲染。

动画引擎由两个类组成，动画对应于AnimationState类，该类中存储着动画的名字和动画的帧:List<Frame>。而Frame中又保存着对应每一帧的图片和这一帧持续的时间的长短。这些动画被添加到状态机StateMachine中形成动画转换表，是HashMap<StateTrigger, String>结构，这个结构中的StateTrigger保存着一个动画的名字以及动画转换的触发器，HashMap中的值就是需要转换到的那个状态。trigger方法提供了便捷索引这个转换表并执行对应的转换的功能。

对话系统由DialogSystem负责，该系统中存储有count和dialogs以及两个对话对象的引用。在对话内容开启时，用户按下N键进行下一句对话，当所有对话完成后，对话框不再更新，直到下一次对话的开启。对话的状态每一帧都会检查一次。

场景系统：

场景是Scene，场景中定义了游戏物品List<GameObject>，提供了创建游戏物品的方法createGameObject，添加游戏物品的方法addGameObjectToScene和获取游戏物品的方法getGameObjectByUID、getGameObjectWith、getGameObjectByName。提供了加载场景的方法load和保存场景的方法save，具体加载和保存的路径由场景加载器指定。值得注意的是，由于引入了物理引擎，不能在物理引擎更新的过程中动态的在游戏场景类添加游戏物品，所以需要另外开一个游戏物品缓冲链表，如果在物理引擎更新的过程中需要添加的物品就先添加到这里来，然后再物理引擎更新完成之后再将游戏物品缓冲链表中的所有游戏物品添加到场景的游戏物品链表中。

场景加载器是SceneInitializer，loadResources方法用于加载资源，init方法用于将用户自定义的游戏对象添加到场景中，实现场景的初始化，而imgui方法可以被用户重写，以实现用户自定义的界面，比如游戏进行都某个位置时弹出一个框，或者弹出游戏人物对话等等。

实用工具类：

在设置中使用glfwInit来初始化初始化GLFW框架，使用glfwGetVideoMode来获取屏幕的分辨率，并将其设置到Settings的成员RESOLUTION\_WIDTH和RESOLUTION\_HEIGHT中，这样在游戏引擎的其他地方就可以轻松的获取这些信息，修改时也只需要修改这一处就可以了，如果用户愿意的话可以实现一个设置界面来修改这些存在于Settings中的成员变量来实现游戏的相关设置。

旋转运算器指的是JMath，使用JMath中的rotatePoints方法可以将一个点绕着另外一个点旋转某一个角度，这样就可以实现Tranform中的rotation的设置。同时支持场景中游戏物品的旋转功能。

资源池指AssetPool，这个池子拥有Map<String, Shader>着色器池，Map<String, Texture>纹理池，Map<String, Spritesheet>精灵表池，Map<String, Sound>声音池，这些都可以通过getXXX的方法来获取，这个方法接收的是资源的路径，返回的是对应的资源，这样就实现了资源的统一管理，并且使用了资源池后不会出现多余的资源，着色器不会被多次编译，永远只有一个，声音也不会被多次缓存到内存中。

以上就是对一些系统组件的实现细节的介绍。具体的细节可以参考配套的代码。下面给出一些对于关卡编辑器操作的按键：

鼠标中间按住可以拖动场景

鼠标滑轮滚动可以缩放场景

鼠标左键点击选择场景中的一个物品

鼠标右键按住拖动框选多个物品

Ctrl+D复制物品

Delete删除物品

PageUp/PageDown上下移动物品的层数

方向键可以移动选中的物品，一次移动一格

按住Shift和方向键可以移动选中的物品，一次移动十分之一格

Ctrl+G切换操作箭头，在物品移动箭头和缩放箭头之间切换

Ctrl+R重置摄像机的位置到默认位置

N键进行下一句对话

**（4）技术讨论**

这里讨论一下软件的不足之处：

（1）、游戏引擎，关卡编辑器和游戏本身没有实现分离，如果分离的话可以做成一个游戏引擎库。然后用这个库开发一个关卡编辑器，再用关卡编辑器开发一个游戏。这样做成三个软件就可以比较好的实现低耦合。目前这个游戏引擎还是把三个部分合在一起的，耦合度比较高。

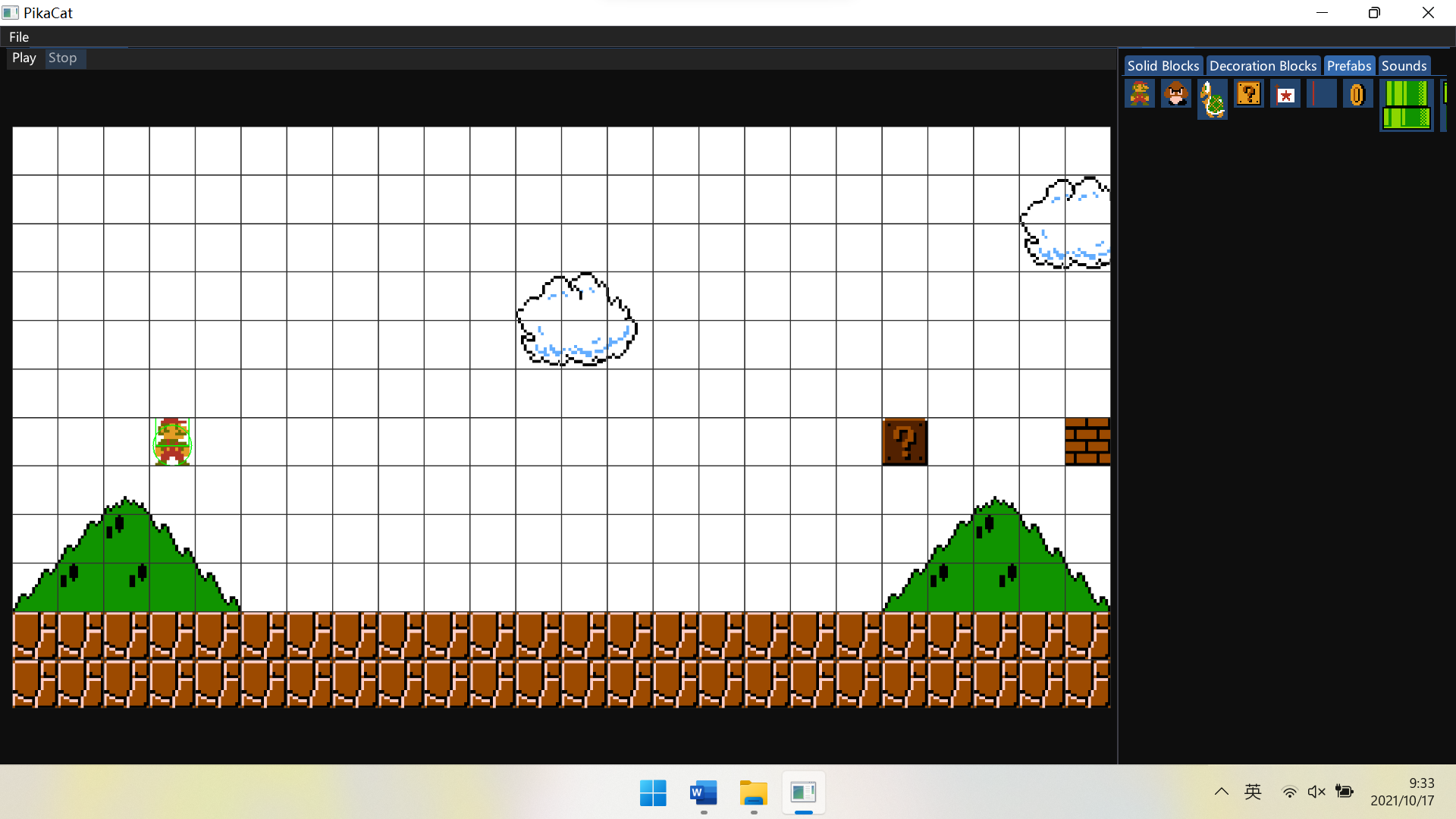
（2）、不能在关卡编辑器中动态加载资源，只能通过场景加载器利用代码的方式加载需要的资源到资源加载窗口中，然后才能利用资源加载窗口将物品拖动到场景中放置。

（3）、不支持脚本语言，例如Lua等，场景内的怪物的AI和游戏对象碰撞的处理需要使用Java代码编写，不能只使用关卡编辑器就能完成绝大部分功能。

（4）、性能不太好，GPU占用率和CPU占用率均非常高，如果需要追求性能需要使用C++进行开发，因为使用Java通过JNI套了很多层，而且还要通过JVM，性能并不高。

**（5）运行截图**

**关卡编辑器**



游戏运行

