3D 引擎开发笔记

Zhang Yf[1]

2021 年 11 月 7 日

[1]Email: 759094438@qq.com

目录

第一部分

序言

序言

0.1 新的开始

从本科时期开始接触代码后，我便一直希望能够进行自己的项目开发，日常也喜欢写一些个人小程序自娱自乐，因为一直对游戏较为感兴趣，因此初期希望研究并开发类似 Unity 的游戏引擎。困于本科时期知识基础的薄弱，以及相关开发经验的缺乏，四年间都只做过一些较小的个人项目，知识体系尚未形成一个较为系统的架构, 没有将个人的思想融入到项目中实现。

研究生阶段，为了能够对读研生涯进行记录，我开始了这个项目的设计。但是由于毕业与工作的压力，之前的文章写了部分就暂时搁浅了，而现在终于有时间来继续推进。我的目标是将此项目作为一个完整的系统，搜集资料后按照个人的想法进行设计和实现，不仅能够锻炼提升个人水平，还能够锻炼阅读与其他能力。同时，这个系列的文章不仅是我个人学习过程的笔记，也将作为后续编写类似程序的相关参考，并在这之中提出一些我个人的想法来抛砖引玉，得到他人的指导或同好的学习与分享。最后希望通过此项目对相关领域做出一些微薄的贡献。

本文使用 LaTeX 编写。

0.2 初步的写作计划

此项目聚焦于跨平台 3D 引擎的设计开发，从零开始搭建一个 3D 引擎用于渲染 3D 场景，以及支持脚本及 AI 的场景内物体互动，使其具有成为仿真环境的可能性。本项目模仿游戏引擎进行模块划分，一个游戏引擎由许多模块组成，每个模块具有相对独立的功能，参考《游戏引擎架构》的内容，本文章将按照以下几个部分设计介绍。当然也有部分章节可能脱离主线，介绍一些额外的内容。

0.2.1 基础系统

这个部分用于介绍引擎底层系统的设计，作为学习的必要过程，本项目中将跟随开发进度不断修改引擎底层结构的设计。许多基础的数据结构与算法也会出现在本节。

0.2.2 数学与几何

随后的渲染与物理部分，都与数学紧密相关。数学构成了渲染与物理模块的基石。本节主要介绍一些基本的数学内容，作为后续部分的基础。

0.2.3 渲染系统

为了能够展示场景的运行效果，需要有直观的方式展现内容，3D 渲染就是最好的方式。这个部分主要介绍渲染的相关内容，包括一个软渲染器以及对应的 GPU 版本渲染器。

0.2.4 物理系统

本节展示一个简单的物理仿真模块，用于刚体碰撞与刚体动力学仿真。为了引擎具有物理仿真的基础功能，也为了更好的表达引擎之中的交互，引入了物理模块作为底层。

0.2.5 场景管理系统

这个部分用于介绍较为大型的场景优化所需的场景管理模块的相关内容。

0.2.6 脚本及 AI 系统

本节主要介绍如何集成一个脚本系统，与场景部分相结合。由脚本系统的功能扩展 AI 的功能。

第二部分

基础模块

第一章　从零开始

1.1　第一个简单项目

作为笔记的第一部分，本节主要介绍从零开始搭建一个小型项目的过程。大部分的 C++ 项目都需要选择（虽然其他项目也是，但是没有 C++ 这么多的选项）一个基本的开发环境，而 CMake 是其中一个流行的选择。从 github 看去，很多国内外大型的开源项目都选择了 CMake 进行项目的配置和管理。前几年我也是 CMake 的推荐者，但学习过 CMake 的我仍然觉得 CMake 过于死板，需要学习额外的 DSL（Domain Specific Language），无法轻易更改编译过程。经过查找，我发现了一个国产的开源项目 xmake[1]，可以用 Lua 编写项目配置文件，符合 C++ 编译过程的直觉，且可以通过 xmake 避免复杂繁琐的外部库安装配置过程，让人感觉很舒心。最终本项目决定使用 xmake 作为项目配置软件。

当然，本项目从熟悉的 Hello World 开始，一步步搭建起项目框架。首先是函数主体：

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

Engine/main.cpp

这个就是简单的标准的 Hello World，通过 xmake 可以简单的配置项目：

[1] https://xmake.io

```lua
target("hello_world")
    set_kind("binary")
    add_files("Engine/*.cpp")
```

xmake.lua

在命令行输入xmake后，就可以自动编译项目，输入xmake r name后，就可以运行对应项目，可以进行快速的项目编译与运行测试。

在后续章节中，若是对应的模块代码过于冗长，我将使用伪代码的形式提供运行的主要流程，代码可以在仓库中查看。

1.2　内存检测模块

作为第一个编写的模块，我选择了内存检测的模块。该模块的目的是提供一个跨平台，线程安全的内存泄漏检测器。参考了 Vorbrodt 的 Blog[2]，对其添加了线程安全锁后，实现了一个小型的内存检测器。由于通过宏定义完成了内存检测器的行数输出，所以该检测器只能用于普通的 new 和 delete，对于 placement new 就无力处理了。若想要不用宏定义也可以获取代码调用位置，可能就只能等待 C++20 的 std::source_location 的功能了。

该内存检测模块主要通过检测 new 与 delete 的配对问题来检测内存泄漏。一次 new 对应一次 delete，一次 new[] 对应 delete[]，避免错配。该模块使用了 set 来存储信息，在 new 的过程中插入信息，delete 的过程中检测并删除对应 new 的信息，达到匹配的效果。为了能够在程序退出时自动输出内存泄漏检查信息，定义了一个 *dump_all* 的全局变量，当程序退出时，该变量自动析构，执行最后的内存泄漏检测，并输出结果。

由于实际的代码太长，仅在此记录主要的算法流程，包括初始化函数 [1]，new 函数 [2]，delete 函数 [3]，退出检查函数 [4]。

Algorithm 1 Init Records

procedure INIT RECORDS(*size*)
    Initialize Static Memory_Record Set
    Initialize Static Delete_Type_Mismatch List
    Initialize Static Delete_Too_Much List

[2] https://vorbrodt.blog/2021/05/27/how-to-detect-memory-leaks/

Algorithm 2 New

```
procedure NEW(size)                                   ▷ new function
    malloc a Memory Block
    Insert a Memory Block Info in Memory Record Set
    return Memory Block Pointer
```

Algorithm 3 Delete

```
                                                      ▷ delete function
procedure DELETE(ptr)
    result = Find ptr in Memory Record Set
    if result = True then
        if deletetypematch then
            Remove Pointer Info in Memory Record Set
        else
            Record Info in Delete Type Mismatch List
    else
        Record Info in Delete Too Much List
```

Algorithm 4 ExitCheck

```
procedure EXITCHECK(ptr)
    Find Leak in Memory Record Set and Dump Info
    Find Mismatch in Delete Type Mismatch List and Dump Info
    Find Mismatch in Delete Too Much List and Dump Info
```

关于内存检测模块，后续仍有很多改进空间，首先需要的就是移除宏定义对 new 和 delete 的限制，使其能够运行调用 placement 的 new 与 delete 版本，以及可以在此添加自定义的内存管理器，进行自定义内存分配策略。

1.3　打开第一个窗口

当然作为一个 3D 程序，需要通过显示器才能展示我们的工作成果，而在现代操作系统中，必须打开窗口才能开始展示东西，就和其他软件一样。为了保证本项目的跨平台能力，需要尽可能一致的为不同平台提供相同的封装，在这里本项目选择了常用的 GLFW 库[3]。

[3] https://www.glfw.org/

暂时本项目只使用 OpenGL 进行渲染，根据官网的教程，可以很容易写出第一个窗口程序：

```
#include "Memory/MemoryCheck.h"

#define GLFW_INCLUDE_NONE
#include <GLFW/glfw3.h>
#include <glad/glad.h>

#include <iostream>

void processInput(GLFWwindow *window);

int main() {
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_RESIZABLE, false);
#if defined(RK_MACOS)
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    GLFWwindow* window = glfwCreateWindow(1280, 720, "Rocket", NULL, NULL);
    if (window == NULL) {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    while (!glfwWindowShouldClose(window)) {
        processInput(window);

        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
```

```
39
40          glfwSwapBuffers(window);
41          glfwPollEvents();
42      }
43
44      glfwDestroyWindow(window);
45      glfwTerminate();
46
47      return 0;
48  }
49
50  void processInput(GLFWwindow *window)
51  {
52      if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
53          glfwSetWindowShouldClose(window, true);
54  }
```

Engine/glfw.cpp

之后开展的渲染学习项目，将从这段程序开始扩展，并接入软渲染器来实现场景初步渲染的能力。

1.4　日志输出

1.4.1　为什么需要日志输出

在程序运行过程中，除了编译器带给我们的单步调试功能，开发过程中也需要额外的信息输出方式，用于快速检查程序输出内容是否正常以及快速定位可能存在 bug 的代码位置。为了达到这个目的，本节主要介绍日志输出的部分。

1.4.2　对 spdlog 进行封装

日志输出模块需要快速，可以按照不同的日志等级输出信息，保证多线程安全，能够输出到日志文件等功能。经过挑选，本项目采用了 spdlog 作为日志输出的底层，对其进行了简单的封装。主要包括 *Log.h*[30] 与 *Log.cpp*[66] 两个文件。在程序运行开始需要初始化 Log，在项目终止时需要终止日志输出（虽然可以依靠自动析构，但是显式调用可以使流程更加清晰）。利用宏定义，可以快速添加新的日志分类。

```cpp
#pragma once
#ifdef RK_CONSOLE_LOG
#include <memory>
#include <unordered_map>
#include <spdlog/spdlog.h>

#define INIT_LOG_CHANNEL(x) s_##x##_logger_ = spdlog::stdout_color_mt(#x);\
    SetLevel(level, s_##x##_logger_.get());
#define END_LOG_CHANNEL(x) s_##x##_logger_.reset();
#define DECLARE_LOG_CHANNEL(x) \
    public:\
        inline static spdlog::logger* Get##x##Logger() { \
            return s_##x##_logger_.get(); } \
    private:\
        static std::shared_ptr<spdlog::logger> s_##x##_logger_;
#define IMPLEMENT_LOG_CHANNEL(x) std::shared_ptr<spdlog::logger>\
    Rocket::Log::s_##x##_logger_;
#endif

namespace Rocket {
    enum class LogLevel {
        TRACE = 0, INFO, WARN, ERR, CRITICAL,
    };

    class Log {
    public:
        static void Init(LogLevel level = LogLevel::TRACE);
        static void End();
#ifdef RK_CONSOLE_LOG
        // 声明Log输出频道
        DECLARE_LOG_CHANNEL(Core);
        DECLARE_LOG_CHANNEL(Window);
        DECLARE_LOG_CHANNEL(Render);
        DECLARE_LOG_CHANNEL(Event);
        DECLARE_LOG_CHANNEL(File);
        DECLARE_LOG_CHANNEL(Audio);
        DECLARE_LOG_CHANNEL(App);
#endif
    };
} // namespace Rocket
```

```
41
42 #ifdef RK_CONSOLE_LOG
43
44 #define RK_CRITICAL(x, ...) \
45     do{::Rocket::Log::Get##x##Logger()->critical(__VA_ARGS__);}while(0);
46 #define RK_ERROR(x, ...) \
47     do{::Rocket::Log::Get##x##Logger()->error(__VA_ARGS__);}while(0);
48 #define RK_WARN(x, ...) \
49     do{::Rocket::Log::Get##x##Logger()->warn(__VA_ARGS__);}while(0);
50 #define RK_INFO(x, ...) \
51     do{::Rocket::Log::Get##x##Logger()->info(__VA_ARGS__);}while(0);
52 #define RK_TRACE(x, ...) \
53     do{::Rocket::Log::Get##x##Logger()->trace(__VA_ARGS__);}while(0);
54
55 #else
56
57 #define RK_CRITICAL(x, ...)
58 #define RK_ERROR(x, ...)
59 #define RK_WARN(x, ...)
60 #define RK_INFO(x, ...)
61 #define RK_TRACE(x, ...)
62
63 #endif
```

Engine/Log/Log.h

```
1  #include "Log/Log.h"
2
3  #ifdef RK_CONSOLE_LOG
4
5  #define SPDLOG_FMT_EXTERNAL
6  #include <spdlog/spdlog.h>
7  #include <spdlog/async.h>
8  #include <spdlog/sinks/stdout_color_sinks.h>
9  #include <spdlog/sinks/basic_file_sink.h>
10 #include <spdlog/fmt/ostr.h>
11 #include <spdlog/fmt/bin_to_hex.h>
12 #include <spdlog/fmt/chrono.h>
13 #include <spdlog/fmt/fmt.h>
14
```

```
15  IMPLEMENT_LOG_CHANNEL(Core);
16  IMPLEMENT_LOG_CHANNEL(Window);
17  IMPLEMENT_LOG_CHANNEL(Render);
18  IMPLEMENT_LOG_CHANNEL(Event);
19  IMPLEMENT_LOG_CHANNEL(File);
20  IMPLEMENT_LOG_CHANNEL(Audio);
21  IMPLEMENT_LOG_CHANNEL(App);
22  #endif
23
24  namespace Rocket {
25  #ifdef RK_CONSOLE_LOG
26      static void SetLevel(LogLevel level, spdlog::logger* logger) {
27          switch(level) {
28          case LogLevel::TRACE:
29              logger->set_level(spdlog::level::trace); break;
30          case LogLevel::INFO:
31              logger->set_level(spdlog::level::info); break;
32          case LogLevel::WARN:
33              logger->set_level(spdlog::level::warn); break;
34          case LogLevel::ERR:
35              logger->set_level(spdlog::level::err); break;
36          case LogLevel::CRITICAL:
37              logger->set_level(spdlog::level::critical); break;
38          }
39      }
40
41      void Log::Init(LogLevel level) {
42          spdlog::set_pattern("%^[%l%$][%T][%n] %v%$");
43          INIT_LOG_CHANNEL(Core);
44          INIT_LOG_CHANNEL(Window);
45          INIT_LOG_CHANNEL(Render);
46          INIT_LOG_CHANNEL(Event);
47          INIT_LOG_CHANNEL(File);
48          INIT_LOG_CHANNEL(Audio);
49          INIT_LOG_CHANNEL(App);
50      }
51
52      void Log::End() {
53          END_LOG_CHANNEL(Core);
54          END_LOG_CHANNEL(Window);
```

```
55          END_LOG_CHANNEL(Render);
56          END_LOG_CHANNEL(Event);
57          END_LOG_CHANNEL(File);
58          END_LOG_CHANNEL(Audio);
59          END_LOG_CHANNEL(App);
60      }
61  #else
62      void Log::Init(LogLevel level) {}
63      void Log::End() {}
64  #endif
65  } // namespace Rocket
```

Engine/Log/Log.cpp

1.5　模块划分

为了更加明确各个部分的功能，这个小节主要描述模块通用的接口，不同的功能根据具体模块进行扩展。

1.5.1　IRuntimeModule

```
1   #pragma once
2   #include "Core/Declare.h"
3   #include "Utils/TimeStep.h"
4
5   #include <string>
6   #include <ostream>
7
8   namespace Rocket {
9       Interface IRuntimeModule {
10      public:
11          virtual ~IRuntimeModule() = default;
12
13          // Return == 0 : everything OK
14          // Return != 0 : something wrong
15          [[nodiscard]] virtual int Initialize() = 0;
16          virtual void Finalize() = 0;
17          virtual void Tick(TimeStep step) = 0;
```

```
18          // For Debug
19          [[nodiscard]] virtual inline std::string ToString() const { return
                GetName(); }
20      protected:
21          [[nodiscard]] virtual inline const char* GetName() const = 0;
22      };
23
24      inline std::ostream& operator << (std::ostream& os, const
            IRuntimeModule& r) {
25          return os << r.ToString();
26      }
27  }
28
29  #define RUNTIME_MODULE_TYPE(type) virtual const char* GetName() const
        override { return #type; }
```

Engine/Log/Log.h

模块运行的流程如下所示：

Algorithm 5 Main Loop

procedure MAIN LOOP(*size*)
    Init Modules
    Init Timer
    while (**do***True*)
        Update Timer
        Tick Module
    Finalize Modules

第二章　基本算法

本章主要介绍一些可能用在引擎中的基本数据结构及基本算法，包括排序和查找算法。本书的算法主要参考算法第四版，所以很多算法和数据结构的设计会比较类似 Java 的代码。

2.1　算法分支-基本数据结构

本节主要介绍一些常用的基本数据结构，包括基于数组与基于链表的，这些数据结构共同构成后续算法部分的基石。本节所描述的数据结构不保证多线程安全，若需要多线程，则需要后续的修改。

2.1.1　Bag

Bag 指的是只能添加不能取出，所有数据的排列按照放入数据的顺序进行。

2.2　算法分支-排序算法

2.3　算法分支-查找算法

第三章　设计模式

设计模式主要针对已经存在的问题，抽象解决方法。

3.1　Iterator 模式

简介

3.2　Adapter 模式

简介

3.3　Template 模式

简介

3.4　Factory 模式

简介

3.5　Singleton 模式

简介

3.6　Prototype 模式

简介

3.7　Builder 模式

简介

3.8　Abstract Factory 模式

简介

3.9　Bridge 模式

简介

3.10　Strategy 模式

简介

3.11　Composite 模式

简介

3.12　Decorator 模式

简介

3.13　Visitor 模式

简介

3.14　Chain of Responsibility 模式

简介

3.15　Facade 模式

简介

3.16　Mediator 模式

简介

3.17　Observer 模式

简介

3.18　Memento 模式

简介

3.19　State 模式

简介

3.20　Flyweight 模式

简介

3.21　Proxy 模式

简介

3.22　Command 模式

简介

3.23　Interpreter 模式

简介

第四章　渲染

附录 A　附录

后记