PPT制作+汇报（5）	实验报告撰写（25）	创新性（10）	实验基本要求（60）			小组平均得分
			二维图形绘制与变换（20）	三维图形绘制与变换（20）	投影变换（20）	
5	19	8	20	20	20	92

计算机图形学 第一次实验 实验报告

1. 简介

- 小组成员
  - 俞贤皓、邓人嘉、付家齐、肖斌、许一涵、陈锦璇
- 项目Github仓库
  - Boundless Jovial Thriving Utopia Game Engine. BJTU Game Engine, for short.
- 因为这学期有图形学课，所以就想搭一个小型游戏引擎。因此，我们基于本小组成员一起搭建的BJTU Game Engine进行这次实验
- 本实验报告中也包含了搭建BJTU Game Engine的内容

2. 实验目的

- 进一步理解和掌握二维、三维几何变换的基本原理；理解和掌握视图 变换、投影变换的基本原理
- 搭建一个游戏引擎的雏形，需要包含实时渲染模块

3. 实验环境

- 操作系统
  - Windows：俞贤皓、邓人嘉、付家齐、陈锦璇
  - Linux：俞贤皓
  - MacOS：肖斌、许一涵
- 项目依赖
  - C++编译器：GCC 13 或 MSVC (Visual Studio 2022)
  - CMake >= 3.19
  - Python >= 3.11

4. 实验原理与实验内容

- 本节共分为五个部分
  - 构建系统
  - 目录结构
  - 游戏引擎与渲染模块
  - 实验内容（导入图形、变换图形、视图变换等）
  - 实验步骤
- 总而言之，本节主要为 **设计思路和开发思路**，也可以作为项目的 **开发者文档。**

4.1 构建系统

- BJTU Game Engine
  - 使用 **C++** 作为核心开发语言，使用C++17标准
  - 使用 **OpenGL** 作为图形库
  - 使用 CMake 作为构建工具：跨平台支持
  - 使用 Python 作为预处理脚本：给GLSL引入#include预处理指令
  - 使用 GLSL 作为着色器语言
  - C++的所有第三方库均以源码形式导入项目，并在本地从零编译
- 第三方库
  - glfw：窗口管理、用户输入
  - glad：OpenGL API
  - glm：数学库
  - assimp：加载模型
  - stb：加载图像
- 构建流程
  - Python和CMake对GLSL进行预处理
    - 注：这里是为了给GLSL带来 `#include` 这个预处理命令，原生OpenGL不提供这个功能
  - CMake编译第三方库
  - CMake编译BJTUGE的C++代码
  - 得到可执行文件

4.2 目录结构

- BJTU Game Engine的目录结构，参考自GAMES104的 Piccolo引擎
- `./engine` 存放主要代码
  - `./engine/3rdparty` 存放第三方库的源码
  - `./engine/asset` 存放所需的所有资源（图片、模型、材质）
  - `./engine/shader` 存放所有着色器代码
  - `./engine/source/runtime` 存放游戏引擎的主要源码
    - `.../main.cpp` 程序入口
    - `.../function/render` 渲染模块
    - `.../function/framework` 主要逻辑模块
    - `.../function/window` 窗口模块
    - `.../function/input` 输入模块
    - `.../function/swap` 交换区，负责逻辑模块和渲染模块的信息交换
- `./scripts` 存放一些脚本，例如GLSL预处理脚本

- `./notes` 存放一些文档和笔记
- `./build` 构建缓存目录
- `./bin` 构建结果目录

4.3 游戏引擎与渲染模块

- BJTUGE共分为两大模块：渲染模块、逻辑模块
  - 这两个模块禁止相互调用，必须通过事件队列的形式进行信息交换
- BJTUGE采用传统面向对象思路进行开发，在逻辑模块使用组件系统。
- 渲染模块
  - 渲染模块的入口为：RenderSystem类
  - RenderSystem管理三个单例
    - RenderResource：负责管理所有渲染资源
    - RenderPipeline：负责管理所有管线
    - RenderCamera：摄像机
  - 渲染模块共有以下四种资源
    - RenderTexture：管理纹理，并封装OpenGL相关操作
    - RenderMesh：管理网格模型，并封装OpenGL相关操作
    - RenderShader：管理着色器，并封装OpenGL相关操作
    - RenderEntity：渲染的最小单位。该类可以递归引用自身，从而形成一颗树形结构，便于管理复杂资源。可以在树形结构的任意节点挂载RenderMesh
  - RenderSystem会把资源从RenderResource中取出，并传入RenderPipeline进行渲染

4.4 实验内容（导入图形、变换图形、视图变换等）

- 在渲染模块的开发中，我们对许多基本操作都进行了封装，使得BJTU Game Engine可以快速实现基本操作
- 导入图形
  - 导入图形需要将资源载入RenderResource，具体来说，有两种方法
  - 第一种方法，在代码中硬编码图形的顶点信息，并将顶点信息载入RenderResource
  - 第二种方法，使用assimp库加载模型信息，并将模型信息载入RenderResource
- 变换图形
  - RenderEntity提供了setModelMatrix方法，可以快速设置当前节点的模型矩阵
  - 使用glm库，得到对应的平移、旋转、缩放、切变、反射矩阵，并用这些矩阵作用于特定RenderEntity
- 视图变换
  - RenderCamera封装了摄像机与视图变换相关的功能
  - 它会根据当前摄像机的坐标、gaze向量、worldup向量，算出视图矩阵
- 投影变换

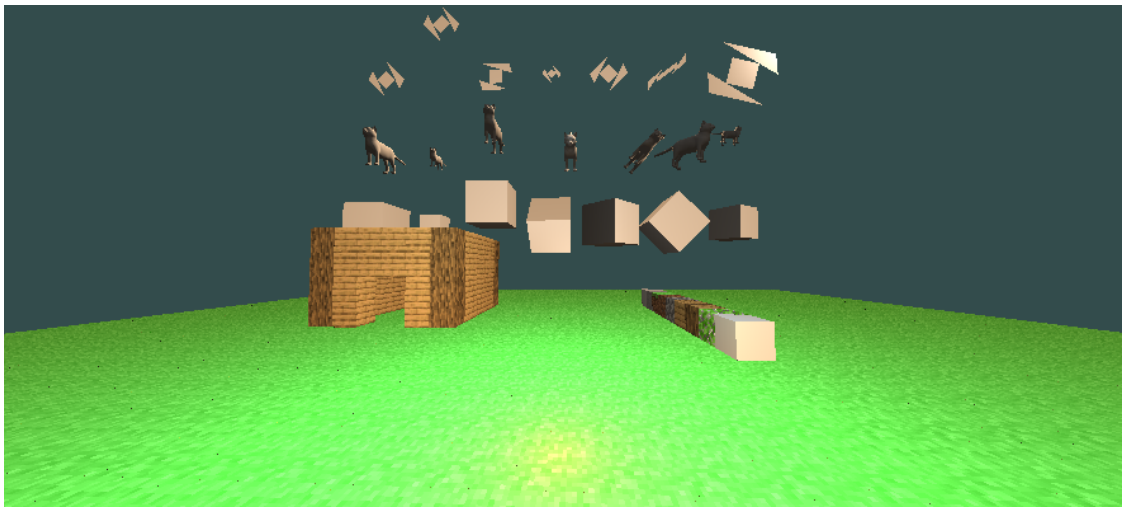
    - RenderCamera同时封装了投影变换相关的功能
    - 它会根据Fov、近平面、远平面等参数，算出透视投影矩阵
    - 它也可以根据六个坐标，算出正交投影矩阵
- 键盘对场景的控制，使用键盘移动摄像机，在透视投影和正交投影之间切换
    - 首先，InputSystem对用户输入进行监听，并得到用户输入状态
    - 接着，RenderCamera会检测InputSystem中的用户输入状态。若用户按下特定按键（例如WASD），则会修改摄像机位置。若用户按下特定组合键（例如Shift+O），则会切换渲染模式（例如在正交投影和透视投影之间切换）
    - 用户也可以使用鼠标，长按右键，拖动摄像机进行旋转

4.5 实验步骤

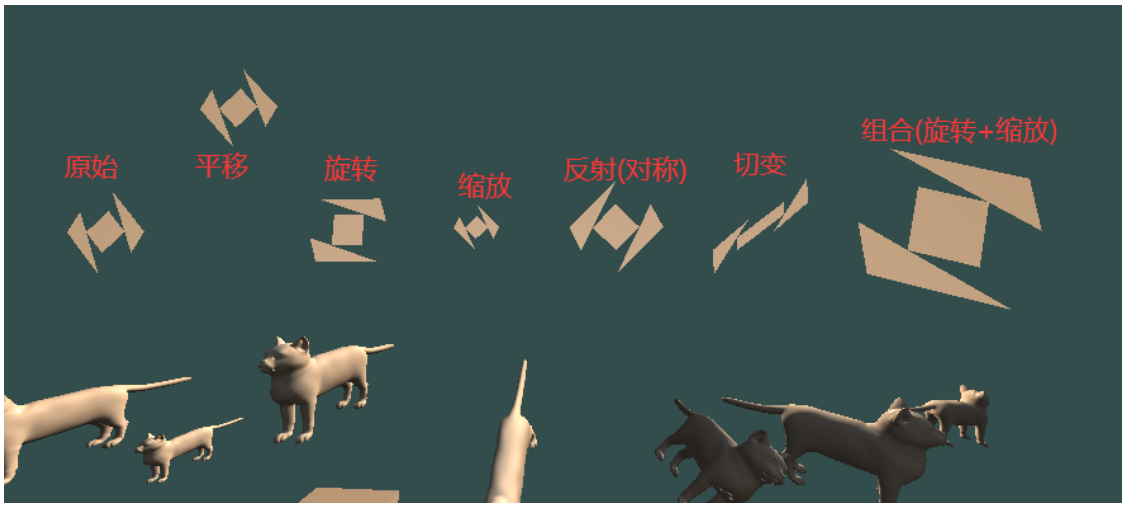
- 完成BJTU Game Engine雏形的开发
- 在RenderResource中导入二维图形或三维图形
- 在RenderResource中对二维图形或三维图形进行变换
- 在RenderPipeline中对二维图形或三维图形进行绘制
- 在InputSystem中添加对特定按键的监听与事件

5. 实验结果

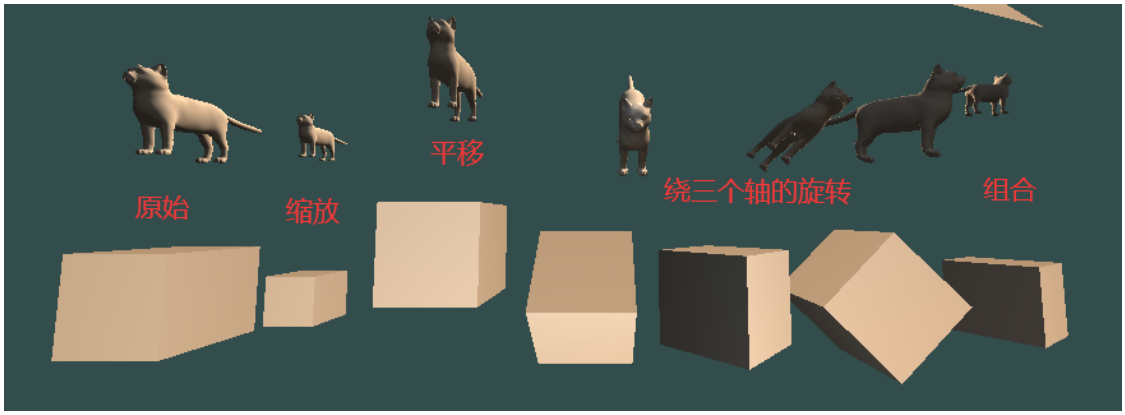
- 实现功能：二维/三维图形渲染、变换、模型导入、纹理映射、光照系统
- 操作方法：
    - WASD控制前后左右移动、QE负责上下移动
    - Shift + O 可以切换视图
- 效果图：
    - 总览：



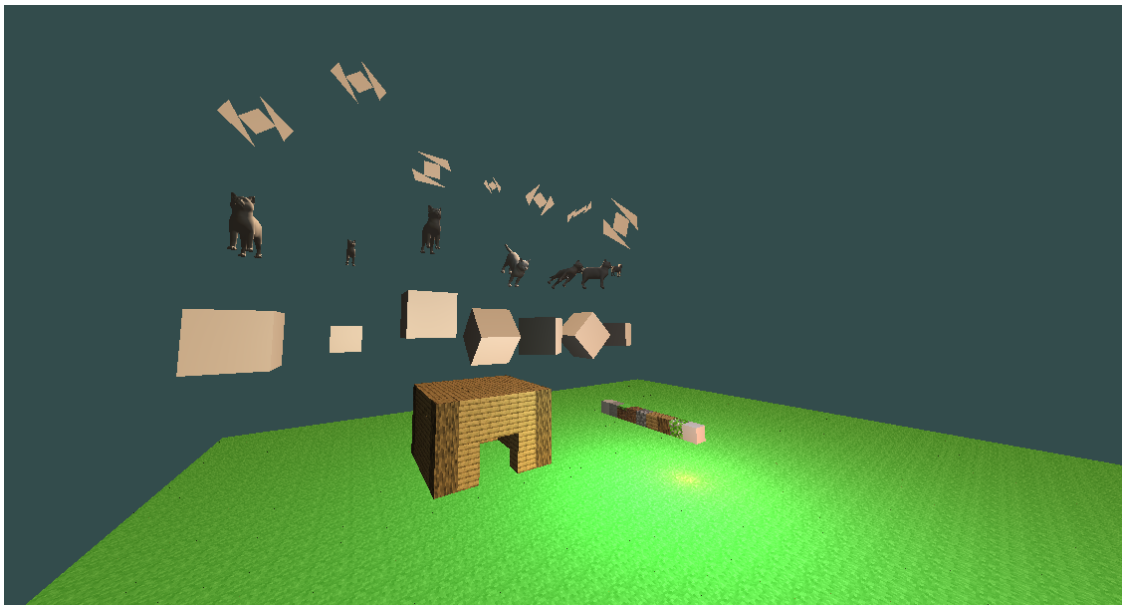
    - 二维图形与变换



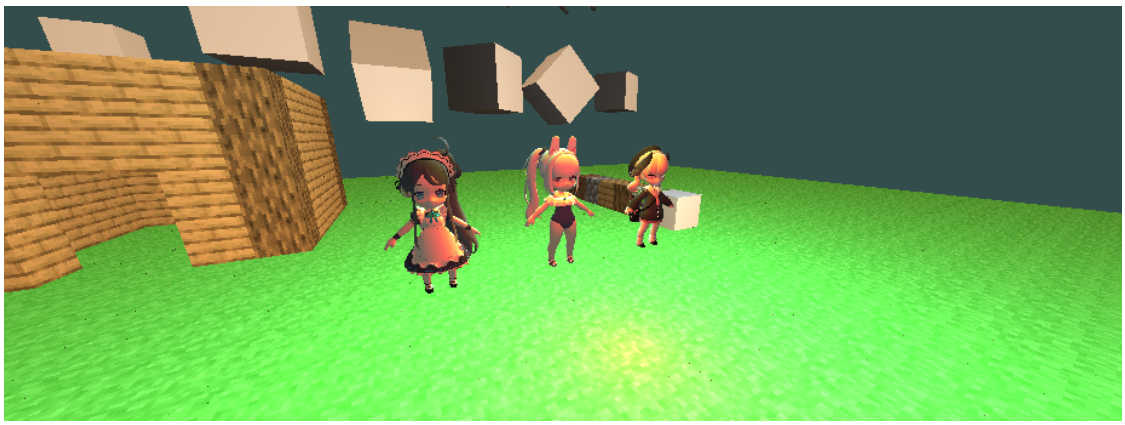
- 三维图形与变换



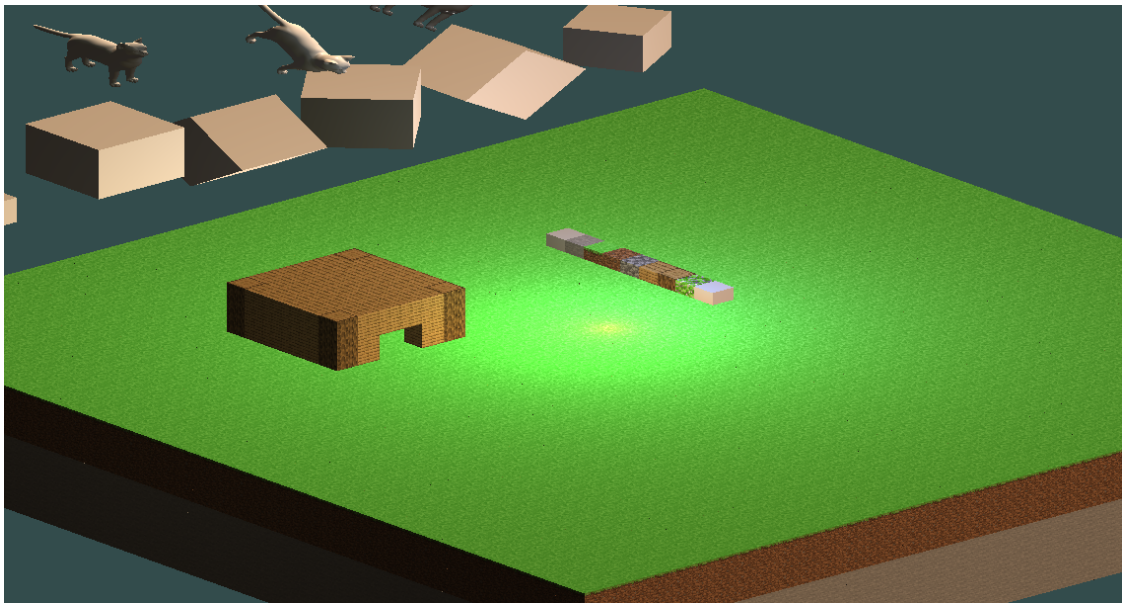
- 用键盘控制视角移动（移动视角）

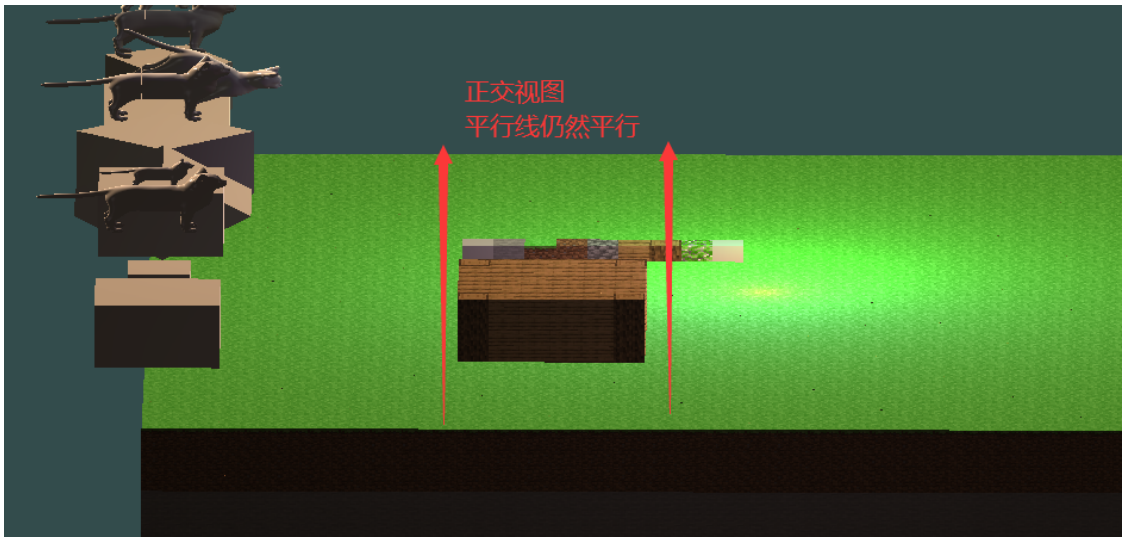


- 用键盘控制场景（是否渲染方块、人物）

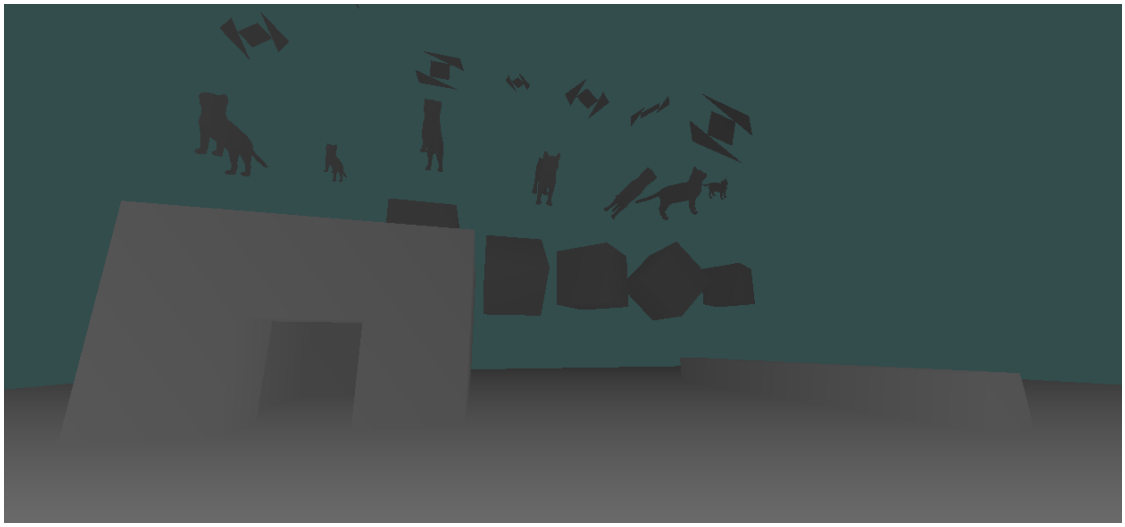


- 正交投影变换





- 按深度渲染场景（按 Shift + P 启动）



6. 实验总结

6.1 遇到的问题

- 问题还蛮多的，这里挑了几个有记录下来并且印象深刻的

6.1.1 MacOS渲染问题

- 问题：MacOS无法唤起窗口
- 原因：
  1. 忘记调用 `glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE)` 这个函数。OpenGL在MacOS上，需要调用这个函数才可以正常唤起窗口
  2. MacOS最高只支持 **OpenGL 4.1**！而OpenGL 4.2在2011年就发布了！苹果你在干什么（逃）
     - 或者OpenGL实际上在很早以前就已经被淘汰了？
- 解决方法：添加函数调用，并修改版本即可

6.1.2 CMake问题

- 问题：CMake编译脚本编译不过。Debug了两个多小时
- 原因：在使用CMake的add_custom_command语句，对字符串进行处理时，多写了一个参数 `VERBATIM`，导致不知道为什么编译错误。
  - ChatGPT4让我加 `VERBATIM` 这个参数，但是删掉后编译就过了（笑
- 解决方法：删掉 `VERBATIM` 这个参数

6.1.3 C++析构函数问题

- 问题：在对Texture进行封装后，发现屏幕一片黑，什么都画不出来。也Debug了两个多小时才解决
- 原因：构造函数使用不当，导致资源被多次析构
  - 析构函数：

```
16      // if (m_channels == 3) {
15      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, m_width, m_he
14      // } else if (m_channels == 4) {
13      //     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_wid
12      // }
11      glGenerateMipmap(GL_TEXTURE_2D);
10
 9      // free resource
 8      stbi_image_free(data);
 7  }
 6
 5  RenderTexture::~RenderTexture() {
 4      std::cout << "!" << std::endl;
 3      glDeleteTextures(1, &m_texture);
 2  }
 1
56  } // namespace BJTUGE
```

  - 错误片段：

```
20
19  void RenderResource::initialize() {
18
17      std::vector<Vertex> vertices = {
16          Vertex{{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 0.0f}, {1.0f, 1.0f}},
15          Vertex{{0.5f, -0.5f, 0.0f}, {0.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
14          Vertex{{-0.5f, -0.5f, 0.0f}, {0.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
13          Vertex{{-0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 0.0f}, {0.0f, 1.0f}},
12      };
11      std::vector<uint32_t> indices = {0, 1, 3, 1, 2, 3};
10
 9      RenderEntity render_entity;
 8      render_entity.m_render_meshes.push_back(RenderMesh{vertices, indices});
 7      render_entity.m_render_textures.push_back(RenderTexture{"./asset/textures/container.jpg", 0});
 6      render_entity.m_render_textures.push_back(RenderTexture{"./asset/textures/mika.png", 1});
 5
 4      m_render_entities_map["basic"] = render_entity;
 3  }
 2
 1  } // namespace BJTUGE
24
```

  - 在上面这个代码中，我将一个RenderTexture传入了push_back函数。

    - 但我这里没有使用指针，导致RenderTexture在该函数内就已经被析构了。
    - 同时，RenderEntity在图中第3行也被析构了一次。
    - 这就导致，一个纹理在使用前就已经被析构了两次！所以调用OpenGL API进行使用的时候，显存中已经没有纹理的对应数据了
- 解决方案：使用指针来管理数据。
    - 另一种解决方案：这里出现问题的原因，是因为我们没有遵循C++的零三五法则，在定义析构函数的时候没有同时定义拷贝构造函数和拷贝赋值函数。如果定义了这两个函数，那么就可以正确管理资源。

6.1.4 glGetProgramiv参数错误

- 问题：在Windows和Linux系统中，shader代码可以正常编译。但是在MacOS中，shader不能正常编译。
- 原因：
    - 我不小心把参数写错了，具体代码如下：
    - 错误：
    ```
    glGetProgramiv(program, GL_COMPILE_STATUS, &success);
    ```
    - 正确：
    ```
    glGetProgramiv(program, GL_LINK_STATUS, &success);
    ```
    - 看来还是MacOS实现上严谨一些。或者也可以说，Windows和Linux的OpenGL实现容错性更高。
- 解决方案：将参数改为正确的

6.1.5 旋转再平移

- 问题：旋转模型后，发现平移方向不对
- 原因：
    - 想让一个模型先绕x轴旋转90°，再按y轴向上平移几个单位
    - 但如果仍然挪y轴，会发现模型实际中会按照世界坐标系的负z轴移动
    - 这是因为平移是按照局部坐标系移动的，所以旋转后再平移，需要按照局部坐标系平移
- 解决方案：
    - 如果要：让一个模型先绕x轴旋转90°，再按 **y轴** 向上平移 x 个单位（世界坐标系）
    - 那么应该：让一个模型先绕x轴旋转90°，再按 **z轴** 向上平移 x 个单位（局部坐标系）
    - 习惯了这种思考方式后，发现还是蛮直观的
- 那为什么不能先平移再旋转呢？因为这样的话，旋转点就不是原点了，很糟糕。
- 试了下，发现缩放也有这个问题。如果先平移再缩放，实际上平移的单位也会被一起缩放！非常糟糕！
- 所以最符合人类直觉的变换顺序，应该是：缩放、旋转、平移

6.2 实验感想

6.2.1 俞贤皓

终于等到了图形学课！太好了！

第一次接触图形学大概是在高中吧，因为喜欢开发游戏，然后就对着GAMES101把图形学学了一遍。以后也准备去图形学工业界，从事游戏引擎的开发。虽然目前AI前景比CG好非常多，但我还是觉得应该学习自己喜欢的东西。所以这学期就准备借助图形学大作业的机会，和朋友一起造一个自研游戏引擎！希望可以成为目前为止我所开发过最复杂最庞大的一个项目。

最开始考虑技术路线的时候，准备使用Vulkan等现代图形库。但学习Vulkan之后，发现难度真的超乎想象，大概需要一两个月才能熟练使用，而OpenGL我已经基本掌握了。后来想了想，觉得应该把重心放在算法和各种feature上，不应该在图形库上花费太多的时间。所以最后还是使用了OpenGL，希望能把更多时间花在实现各种美轮美奂的算法上。

确定了需要做游戏引擎后，这个学期初就开始学GAMES104了，真的有意思啊！然后就对着GAMES104的知识开始搭游戏引擎了。但实践下来，发现细节真的是非常多，很难设计一个优雅的架构。目前，BJTUGE各个模块之间还是存在比较高的耦合性，属于优雅与屎山的分界点。

因为大学阶段其实也已经搭过两次渲染器了（一次C++与QtOpenGL、一次Typescript与WebGPU），所以基本流程都非常熟悉了。总的来说，我希望能让这次图形学大作业超过我以往的任何一个渲染器吧，把它写进简历里，成为我个人图形学生涯中的新一个里程碑。

但目前来说，这个目标还很遥远，需要继续努力。

6.2.2 邓人嘉

这次实验我的任务是使用OpenGL库来进行二维图形绘制及变换的实验，经过实验我对计算机图形学的基础概念有了更深入的了解。

我首先学习了如何使用OpenGL库来创建基本的二维图形，包括矩形和三角形，我学会了如何通过OpenGL的API来渲染这些顶点到屏幕上。

我还学习了如何实现缩放、反射、切变、旋转和平移五种基本的二维变换。每种变换将图形乘上变换矩阵来实现，我运用矩阵运算和坐标变换的基本原理完成了这些变换。

在学习了单独的变换后，我尝试将不同的变换组合应用到同一个图形上。这部分特别需要注意变换顺序对最终结果的影响，我深刻体会到变换顺序不同也会给图形带来很大的不同。通过实验，我明白了变换矩阵乘法的非交换性质，以及如何通过改变变换顺序来达到预期的视觉效果。

实验总结：最终，我展示了七个图形，包括一个原始图形和六个经过变换的图形（五个单一变换和一个组合变换）。这不仅加深了我对图形学基础的理解，也让我掌握了使用OpenGL进行图形编程的基本技巧。通过这次实验，我认识到了数学在图形编程中的重要性，特别是线性代数的知识如何直接应用于图形变换。此外，实验的实践操作加深了我对计算机图形学理论的理解，并激发了我继续探索更高级图形技术的兴趣。

6.2.3 付家齐

在这次图形学的实验中，我深入学习了三维图形变换的核心概念和实现方法。实验的主要目标是创建七个三维模型（立方体），对它们应用不同的变换，包括平移、缩放、旋转，并最终将它们排成一行展示不同的变换效果。这个过程不仅加深了我对三维变换如何影响物体的空间表现的理解，而且也锻炼了我使用图形库GLM进行编程的技能。

学习重点

1. **三维变换的基础**：通过这次实验，我了解到三维变换包括平移、缩放和旋转。每种变换都可以通过一个4x4的矩阵来表示，这在OpenGL中非常关键。学习如何创建和应用这些矩阵是实验的基础部分。
2. **矩阵乘法的顺序**：实验中一个非常重要的概念是变换的顺序问题。在图形学中，变换的实际应用顺序与代码中的顺序是相反的。这意味着最后一个变换先被应用于物体。理解这一点对于正确设置场景和动画至关重要。
3. **GLM库的应用**：GLM是一个广泛使用的数学库，专门为图形软件提供了数据结构和算法。通过实验，我学习了如何使用GLM提供的函数来创建和操作矩阵，这对我的学习非常有帮助。

实践经验

在编写代码实现变换时，我遇到了一些挑战，特别是在理解如何组合使用不同的变换。开始时，我没有正确理解矩阵乘法的顺序，导致立方体的显示位置和预期有很大的偏差。通过反复试验和查阅资料，我逐渐理解了矩阵乘法的逻辑，并成功地将立方体按照预期排列并应用了正确的变换。

此外，调试过程中我发现即使是小错误也会导致完全不同的渲染结果。例如，我一度忘记更新矩阵变量，结果是变换没有被正确应用。这教会了我在编程中需要更加注意细节，以及如何步步为营地验证每一部分代码的正确性。

总结

这次实验不仅增强了我对图形学理论的理解，也提高了我使用图形库进行编程的实践技能。通过亲自操作和观察三维模型的变换，我更加直观地理解了理论知识的实际应用。我期待在未来的学习中继续探索更多图形学的概念和技术，如光照模型、着色技术和高级渲染技术等。这次实验是我图形学学习旅程中的一个重要里程碑，我相信所学的知识将为我未来的学术和职业生涯打下坚实的基础。

6.2.4 肖斌

首先，对于图形学知识来说，经过本次实验，通过实际编码体会了渲染管线中的每个过程，对整个渲染过程有了更具体的认识与理解，感觉很有意思。

其他方面来说，本次实验中，我们基于 OpenGL API 使用 C++ 开发。由于我们组中 Mac、Linux、Windows 三个平台都有，我们也就被迫遇到了很多不同平台之间的兼容性问题。比如 Mac 已宣布停止对 opengl 的支持，最高只支持到 4.1，以及在材质为空的时候，Mac 与其他平台的渲染结果也不一致。这导致我们额外消耗了大量精力来解决平台差异性带来的问题。cmake 也增加了我们的头疼。

这些“一脉相承”过来的东西（图形api、着色语言、编程语言、编译工具等等）固然有着极为深厚的积淀，然而却同时带着过于沉重的历史包袱。这一点在 cpp 身上尤为明显，“现代cpp”中确实“与时俱进”地引入了很多现代的理念，以及从各个其他语言取经到的概念，然而它必须保证向后兼容，这就使得它变得越来越复杂，即“内卷”一词的本意。它们无法到达一个崭新的层次，无法打破历史的包袱，于是只能在自己的内部变得越来越复杂。

不过好在是总有新旧事物的更迭，一切都在轮回中前进，新兴的 Rust 实现的 wgpu 以及其 WebGPU 规范正在发展之中，简单体验了一下，感到兴奋。大概唯有不安于现状，不得过且过，带着“创造更多美好”的初心不断前行，才能够创造出这些东西。反感国内互联网企业现状，囿于 java、springboot、vue2 等等老旧技术，只有极少企业在尝试新事物，真正地创造新事物。不得不感叹，我们软件人，实乃任重而道远。

> 俞贤皓 注：当时考虑技术路线时，也和肖斌有讨论。Rust和wgpu确实是一个同样优秀的技术方案，但最后还是选择了C++和OpenGL。肖斌是Rust高手，所以之后，我们可能会考虑在BJTUGE中，同时使用C++和Rust进行开发，或者使用Rust对渲染部分进行重构。这太酷了！

6.2.5 许一涵

学习部分

在本次实验中，首先我学习了 Learn OpenGL 教程的入门部分，了解了基本的opengl使用方式。
在学习中，我了解了如何使用opengl是如何在屏幕中渲染二维及三维物体的，我还学习了如何在opengl中对图形进行变换。
此外，MVP变换也是其中的一大学习重点内容，在代码中我通过opengl中封装的lookat函数成功实现了相机视角的变换，并且成功切换了正交投影与透视投影。

实验部分

在本次实验中我主要负责添加键盘对场景的控制，以及切换正交与透视投影。
通过在input_system中添加键盘监听修改状态变量从而控制特定部件的渲染与否。此外，在切换正交投影与透视投影上，通过按钮控制相机类返回正交投影矩阵或者透视投影矩阵，从而控制正交与透视视角。

实验感想

opengl库不同于以往学过的任何库，使用起来的流程较为繁琐，在学习过程中，代码通常使用一个unsigned int表示一些元素与opengl库进行操作，代码的具体顺序通常难以记住，需要根据教程里的代码依葫芦画瓢。
此外，我们还需要自己撰写glsl代码用于指定opengl如何对点进行渲染。通过本次实验，我加深了对MVP变换、正交投影、透视投影以及二三维变换的理解，并且在代码中对他们进行了尝试。

6.2.6 陈锦璇

学习部分：

在本次实验中，通过使用OpenGL进行图形绘制及变换操作，我学习了三维图形的绘制和变换。在实验过程中，我逐步掌握了OpenGL的基本操作和函数调用。最终我绘制了7个图形，分别为原模型、平移、缩小0.5倍、沿x/y/z轴旋转以及组合变换。

实践经验：

一开始我使用了球体进行三维变换，进行到旋转时发现球体无法展示出绕轴旋转的效果，于是将球体模型换成了小猫模型。导入小猫模型后出现了以下三个问题：

1. 模型不在我想要展示的理想初始位置。
2. 由于模型尺寸过大导致无法显示的问题。
3. 导入后的小猫全部脸朝下。

针对上述问题我采取了以下解决方案：

1. 通过平移，我将小猫挪到了理想的初始位置。
2. 将小猫缩小0.005倍，调整到了合适的大小。
3. 通过让小猫沿z轴方向旋转270度，将小猫调整到了正常的站立姿势。

这使得我对于模型的三维变换有了更深刻的认识。

总结：

通过实现了七个不同的图形，我对图形学的理论知识有了更加深入的理解。五个单一变换的实现让我能够熟练地应用平移和旋转等基本操作，而组合变换则进一步拓展了我的思维，让我能够将多种变换结合起来，创造出更加复杂和多样化的图形效果。

---

7. 实验分工

标签	姓名	学号	班级
组长	俞贤皓	21301114	软件2104
组员	邓人嘉	21301032	软件2102
组员	付家齐	21301034	软件2102
组员	肖斌	21301021	软件2101
组员	许一涵	21301172	软件2106
组员	陈锦璇	21301119	软件2105

- 以下内容参考Github仓库的 Commit记录
- 俞贤皓
  - 构建系统搭建（CMake）、渲染模块搭建（设计框架、封装OpenGL、基础结构）、引擎框架搭建、项目管理（文档等）、其他（没写的都是我做的）
- 肖斌
  - 构建系统搭建（justfile）、渲染模块搭建（设计框架、封装OpenGL、光照系统）、GLSL预处理器开发
- 邓人嘉
  - 二维图形绘制、二维变换、二维组合变换
- 付家齐
  - 三维图形绘制、三维变换、三维组合变换（第一种模型）
- 许一涵
  - 键盘对场景的控制、在透视投影和正交投影中切换
- 陈锦璇
  - 三维图形绘制、三维变换、三维组合变换（第二种模型）