

西安交通大学计算机图形学实验文档

基础部分

作者：李昊东

组织：计算机图形学课题组

时间：September 6, 2023



目录

第 1 章 前置实验	2
1.1 实验内容	2
1.2 指导和要求	3
1.2.1 项目管理	3
1.2.2 数值计算	4
1.2.3 日志	5
1.2.4 格式化输出	7
1.2.5 要求	8
1.3 实验结果	8
1.4 提交内容	9
第 2 章 配置 Dandelion 环境	10
2.1 实验内容	10
2.2 指导和要求	10
2.2.1 编译和运行	10
2.2.2 模式与功能	11
2.2.3 要求	13
2.3 提交内容	13
第 3 章 变换矩阵	14
3.1 实验内容	14
3.2 指导和要求	14
3.2.1 齐次坐标和变换矩阵	14
3.2.2 平移、旋转和缩放	14
3.2.3 要求	15
3.3 实验结果	16
第 4 章 透视投影矩阵	17
4.1 实验内容	17
4.2 指导和要求	17
4.2.1 透视投影矩阵	17
4.2.2 要求	18
4.3 实验结果	18
第 5 章 欧拉角到四元数的转换	20
5.1 实验内容	20
5.2 指导和要求	20
5.2.1 欧拉角的弊端	20
5.2.2 轴-角表示法和四元数	21
5.2.3 要求	21
5.3 实验结果	21
5.4 提交和验收	22

附录 A 如何提交实验结果	23
A.1 注册	23
A.2 邮件格式和反馈	23
A.3 实验结果自动检查	24
附录 B 选做实验	25
B.1 选做实验的内容和选择方式	25
B.2 选做实验的考核方式	25
B.3 代码查重与作弊处理	25
B.4 挑战任务	26
参考文献	27

序

大家好！这份文档是各位同学计算机图形学实验的开篇介绍。在这部分实验中，你需要练习使用 Eigen 进行数值计算、使用 spdlog 记录日志等基础技能；并试着编译 Dandelion 3D、为它增加最基本的三维坐标变换功能，从而让场景布局和预览成为可能。在完成基础部分之后，你将得到一个可以加载模型、布置场景的场景编辑器。

这部分文档总共包含四个必做实验和一个选做实验，分别是

- 前置实验：一些基本技能训练。
- 配置 Dandelion 环境：下载并编译 Dandelion 源代码。
- 变换矩阵：将平移、旋转、缩放三种变换转化为相应的变换矩阵。
- 透视投影：根据相机参数计算透视投影矩阵。
- 四元数到欧拉角的转换（选做）：将四元数转换为 ZYX 欧拉角。

最后，我们也会介绍本学期选做实验的路线图，并附上提交实验结果的基本规范。让我们开始吧，画一些有趣的东西！¹²

¹这份文档使用 ElegantBook 模板编写，按 CC BY-NC-SA 4.0 协议发布。

²封面图是 Dandelion 布局模式下，用平移操作拆分一架滑翔机各部件的效果。

第 1 章 前置实验

前置实验（编号 1.0）是正式实验开始前的预备，主要训练基本编程技能，兼有检验编程能力的作用。

1.1 实验内容

还记得线性代数中学过的抛物面吗？

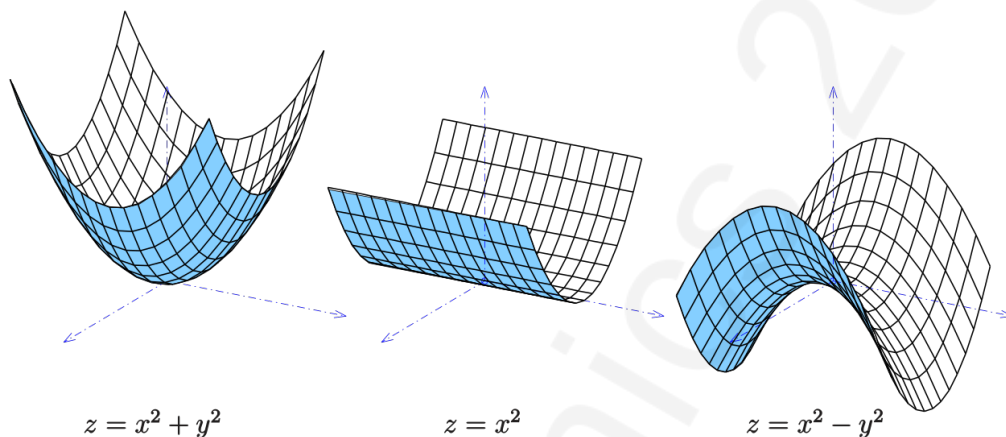


图 1.1: 椭圆抛物面、抛物柱面、双曲抛物面，图片来自 [1]

图 1.1 中的椭圆抛物面其实是更加特化的旋转抛物面，如果将 z 值视为“高度”，我们可以把它看作是下面这个函数的图像：

$$f(x, y) = x^2 + y^2$$

显然， $f(x, y)$ 在 \mathbb{R}^2 上的最小值是 0，并且它的图像始终是向下凹陷的。对于这种函数，我们可以用**牛顿下降法**快速计算它的最小值。牛顿下降法的计算过程是这样的：

1. 选定一个初始点 $x_0 = (x_0, y_0)$ 和下降步长 λ
2. 对于第 k 步的点 $x_k = (x_k, y_k)$ ，计算梯度 $\nabla f(x_k)$ 和导数（Hessian 矩阵）的逆 $H_f^{-1}(x_k)$
3. 迭代 $x_{k+1} = x_k - \lambda H_f^{-1}(x_k) \nabla f(x_k)$
4. 当这次下降的程度小于某个阈值 δ ，即 $\|x_{k+1} - x_k\| < \delta$ 时，停止迭代并认为找到了极小值点 x^*

在前置实验中，你需要根据上面的计算过程实现一个用牛顿下降法求 $f(x, y) = x^2 + y^2$ 极小值的程序，并按照表 1.1 选取参数。

表 1.1: 牛顿下降法参数表，mod 表示取余运算

参数	取值
初始点 x_0	记你的学号为 p ，取 $x_0 = (p \bmod 827, p \bmod 1709)$
步长 λ	0.5
阈值 δ	0.01

1.2 指导和要求

1.2.1 项目管理

当你需要用多个源文件来编译程序时，你就自然而然地得到了一个项目（Project，或译作工程）。CMake 是目前最通用的 C++ 项目管理工具¹，我们也使用 CMake 管理项目。遗憾的是 CMake 的官方文档写得并不好，所以我们推荐一个第三方教程 **CMake Examples**²。如果你没有使用过 CMake，请至少完成它 01-basic 部分的 A-hello-cmake, B-hello-headers, G-compile-flags 三个练习来了解 CMake 到底是什么。

在这个实验中，我们需要使用 Eigen 和 spdlog 两个库，它们都可以使用 CMake 直接引入。请下载我们提供的 *Eigen-3.4.0.zip*, *spdlog-1.11.0.zip* 和 *fmt-9.1.0.zip* 这三个文件，并将其内容直接解压到项目根目录下的 **deps** 文件夹中，形成如下的文件结构：

```
optimizer
├── deps/
│   ├── Eigen/
│   ├── fmt/
│   └── spdlog/
├── ...
├── main.cpp
└── CMakeLists.txt
```

请在以下模板的基础上编写自己的 *CMakeLists.txt*：

```
cmake_minimum_required(VERSION 3.5)
project(optimizer VERSION 0.1)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
set(CMAKE_EXPORT_COMPILE_COMMANDS TRUE)

set(SOURCES
    # 在这里添加其他源文件相对于项目根目录的完整路径和名称
    main.cpp
)

add_executable(${PROJECT_NAME} ${SOURCES})
target_include_directories(${PROJECT_NAME}
    # 如果你的头文件和源文件不在同一个路径下，可以在这里添加 include 搜索路径
    # 第三方库的头文件全部在 deps/ 下，不需要额外添加搜索路径了
    PRIVATE deps
)
target_compile_definitions(${PROJECT_NAME}
```

¹也许并不是最好的，但几乎所有的主流 C++ 项目都提供 CMake 构建方式

²GitHub: <https://github.com/ttroty50/cmake-examples>, Gitee 上也有很多镜像仓库

```
PRIVATE SPDLOG_FMT_EXTERNAL
PRIVATE FMT_HEADER_ONLY
)
```

1.2.2 数值计算

Eigen 是一个 C++ 线性代数库，提供了方便的向量和矩阵运算。用牛顿迭代法求最小值的过程中要计算的矩阵-向量乘积和逆矩阵，都可以直接用 Eigen 完成。要使用 Eigen 中的类型和函数，首先需要引入头文件：

```
// 包含矩阵类型及其基本运算的定义，通常都需要引入
#include <Eigen/Core>
// 包含一些额外的矩阵运算
#include <Eigen/Dense>
```

Eigen 中基本的类型是 `Matrix`，这是一个模板类，它的定义大致是这样的：

```
Matrix<Scalar, RowsAtCompileTime, ColsAtCompileTime, Options>
```

`Scalar` 表示矩阵元素的数据类型，`RowsAtCompileTime` 和 `ColsAtCompileTime` 则表示行数与列数。例如 `float` 类型的三维方阵与 `double` 类型的 4×5 矩阵定义为：

```
Matrix<float, 3, 3>
Matrix<double, 4, 5>
```

你大概会注意到一个问题——这里没有写出最后一个模板参数。那是因为它已经有默认值 `ColumnMajor`，表示这个矩阵是列优先存储的。Eigen 中的向量不是独立类型，只是列数为 1 的矩阵，显然 Eigen 存储的是列向量。

在图形学程序中，我们通常只使用很低维度的矩阵和向量，Eigen 提供了一些方便的别名，一般只用这些别名就足够了。这些别名形如 `[Matrix|Vector]X[i|f|d]`，其中 X 指定维数，i / f / d 分别代表 `int` / `float` / `double` 类型。下面的两个例子分别是二维方阵和三维向量：

```
typedef Matrix<float, 2, 2> Matrix2f;
typedef Matrix<float, 3, 1> Vector3f;
```

Eigen 重载了 `Matrix` 类型的各种运算符，让大多数计算都很符合日常的书写习惯。无论矩阵是列优先还是行优先存储，书写下标时总是先行后列，与平时书面写法一致；不同的是，矩阵（向量）的下标从 0 开始。向量类型则可以用 `x()` / `y()` / `z()` / `w()` 访问各个分量。

```
using Eigen::Matrix3f;
using Eigen::Vector3f;
```

```
// Initialize matrix A as an identity matrix
Matrix3f A = Matrix3f::Identity();
// Access and modify A's element. After assignment, A will be
// 1.0 0.0 0.0
// 0.0 1.0 2.0
// 0.0 0.0 1.0
A(1, 2) = 2.0f;
// Initialize vector x
Vector3f x(4.0f, 5.0f, 6.0f);
// Matrix-vector multiplication, p will be (4.0, 17.0, 6.0)
Vector3f p = A * x;
// Access and modify p's element, After assignment, p will be
// (4.0, 17.0, 5.0)
p.z() = 5.0f;
```

矩阵转置、求逆、求行列式，以及向量点积、叉积等操作都是类方法：

```
Matrix3f A = Matrix3f::Identity();
// Inversion (transpose) of an identity matrix is itself
Matrix3f A_inv = A.inverse();
Matrix3f A_trans = A.transpose();

Vector3f x = Vector3f::UnitX();
Vector3f y = Vector3f::UnitY();
// Dot production of unit x and unit y is 1
float x_dot_y = x.dot(y);
// Cross production of unit x and unit y is unit z (for right-hand coordinate
// system)
Vector3f z = x.cross(y);
```

更多关于 Eigen 的说明请查询官方文档 [Eigen 3 Documentation](https://eigen.tuxfamily.org/doc/3.3/)。

1.2.3 日志

日志可以记录程序的运行过程与结果，既是有力的调试工具也可以作为日后改进程序的参考，初学者也可以通过详细的日志快速理解一个程序的工作流程。可靠、高效率地输出易读的日志是一件比较复杂的事情，和直接使用 `printf` 或 `cout` 有很大的差别。在本课程中，我们使用 `spdlog`³ 库输出日志。

`spdlog` 功能全面而强大，我们并不过多介绍它的种种复杂特性，只使用基本的部分。它设计了两种关键对象：逻辑层面输出日志的 `logger` 和物理层面写入文件（缓冲）的 `sink`。简单来说，`logger` 就像指挥官而 `sink` 就像士兵，我们通过 `logger` 下达“输出日志”的命令，然后 `sink` 负责将要输出的字符串写到终端或者文件。

下面的代码创建了一个输出到终端 (`stdout`) 的 `sink` 和一个输出到文件的 `sink`，并创建一个与它们关联的 `logger`：

³GitHub: <https://github.com/gabime/spdlog>


```

#include <memory>

#include <spdlog/spdlog.h>
#include <spdlog/sinks/stdout_color_sinks.h>
#include <spdlog/sinks/basic_file_sink.h>

int main()
{
    auto console_sink =
    ↪ std::make_shared<spdlog::sinks::stdout_color_sink_st>();
    auto file_sink =
    ↪ std::make_shared<spdlog::sinks::basic_file_sink_st>("my_program.log",
    ↪ true);

    spdlog::logger my_logger("my logger", {console_sink, file_sink});
    my_logger.debug("This is a debug message");
    my_logger.info("Some information during processing");
    my_logger.warn("Warning, something is going wrong");
    my_logger.error("An error occurred");
    my_logger.critical("Critical error, emergency stop");

    return 0;
}

```

这个 `logger` 的名字是 `my logger`，而调用它的 `debug` / `info` / `warn` 等方法，可以将日志信息同时输出到两个 `sink` 对应的文件（缓冲）。如果直接编译运行这段代码，你会看到这样的输出：

```

[2023-07-15 15:01:23.456] [my logger] [info] Some information during
↪ processing
[2023-07-15 15:01:23.456] [my logger] [warn] Warning, something is going wrong
[2023-07-15 15:01:23.456] [my logger] [error] An error occurred
[2023-07-15 15:01:23.456] [my logger] [critical] Critical error, emergency
↪ stop

```

每一条日志里依次是时间、`logger` 名字、日志级别和日志信息。然而我们会注意到，第一条日志没有出现！这是因为 `spdlog` 从低到高有 `trace`, `debug`, `info`, `warn`, `error`, `critical` 六个**日志级别**，默认只会输出 `info` 及以上级别的日志，而不会输出 `trace` 和 `debug` 级别的日志。为了让程序输出更多的日志，我们需要设置日志级别：

```

// set the global log level
spdlog::set_level(spdlog::level::debug);
// set a logger's log level (assume the logger is a shared_ptr)
logger->set_level(spdlog::level::debug);
// set a sink's log level (assume the sink is a shared_ptr)

```

```
sink->set_level(spdlog::level::debug);
```

以刚才的程序为例：我们可以在创建 `sink` 和 `logger` 之前设置全局日志级别，也可以设置 `my_logger` 的日志级别，还可以设置 `console_sink` 或者 `file_sink` 的日志级别。将日志级别设为 `debug` 后，所有 `debug` 级别的日志也会被输出了。

1.2.4 格式化输出

如果日志只能输出一些简单的字符串，那它显然是不怎么方便的，毕竟无论是 `printf` 还是 `cout` 都有通用的输出能力。所幸 `spdlog` 具备比 `printf` 更安全、比 `cout` 更方便的通用输出方法：格式化字符串。下面我们用格式化字符串输出一些变量：

```
// use the helper function to create a logger with only one sink
auto logger = spdlog::stdout_color_sink_st("my_logger");
// some variables
int a = 10;
float pi = 3.1415926;
std::string message = "Hello, world!";
// log a variable with format string
logger->info("a integer: {}", a);
logger->info("my program says: {}", message);
// format floating point number to a fixed point number
logger->info("pi is {}, approximation: {:.2f}", pi, pi);
```

输出结果会是这样的：（因为文档页面宽度有限，输出内容折行显示，实际上每条日志只占一行）

```
[2023-07-15 15:01:23.456] [my logger] [info] a integer: 10
[2023-07-15 15:01:23.456] [my logger] [info] my program says: Hello, world!
[2023-07-15 15:01:23.456] [my logger] [info] pi is 3.1415926, approximation:
↪ 3.14
```

如果你之前使用过 Python 的 f-string，那么想来你会觉得这种写法相当亲切。`spdlog` 的格式化输出能力由 `fmtlib`⁴ 提供，格式化语法说明请参考 <https://fmt.dev/9.1.0/syntax.html>⁵。

`fmtlib` 直接提供了对 C++ 内置类型、大多数 STL 容器以及 C++ 时间类型的格式化输出，想要用日志记录这些变量时，只要在格式串里放个 `{}` 占位即可。但我们会经常用到 `Eigen` 的矩阵和向量，自然也希望在需要的时候将其记录下来。`fmtlib` 并不能直接格式化它们，而总写下标又非常麻烦，我们可以通过自定义 `formatter` 的方式解决这个问题。不过我们并不要求你掌握这部分内容，你可以直接从 `Dandelion` 源代码库中复制 `utils/formatter.hpp` 这个文件到你自己的 CMake Project 中，然后在引入 `spdlog` 头文件之前先引入这个头文件，就可以像输出内置类型那样输出向量和矩阵了。下面的代码是一个例子，详细说明请参考 [开发者文档：formatter.hpp](#)

⁴GitHub: <https://github.com/fmtlib/fmt>

⁵由于兼容最新版 `fmtlib` 10.0.0 的 `spdlog` 1.12.0 在开发 `Dandelion` 时尚未发布，我们没有使用最新的 `fmtlib`

```
#include "formatter.hpp"
#include <spdlog/spdlog.h>
#include <Eigen/Core>

using Eigen::Vector3f;
using Eigen::Matrix3f;

int main()
{
    Vector3f v(1.0f, 2.0f, 3.0f);
    spdlog::info("vector: {:.2f}", v);
    Matrix3f I = Matrix3f::Identity();
    spdlog::info("matrix: {:>5.1f}", I);
    return 0;
}
```

上面的例子应该足够本学期的实验使用，如果你对 spdlog 还有其他的问题，请查找它的官方文档 [spdlog wiki](#)。

1.2.5 要求

前置实验不提供实验框架，你需要自己编写代码。我们要求：

- 按照表 1.1 选择参数。
- 用 CMake 管理项目。
- 用 Eigen 完成向量计算和矩阵求逆。
- 为了保证精度，所有的浮点数都使用 `double` 类型，向量和矩阵也使用后缀为 `d` 的类型。
- 同时用 spdlog 输出日志到终端和 `optimizer.log` 中，以 `debug` 级别记录每一步迭代的结果（从 x_0 到 x^* ），以 `info` 级别记录得到的最小值。所有向量和数值直接输出，不加任何描述。
- 不允许引入任何其他第三方库。

1.3 实验结果

回顾图 1.1 中椭圆抛物面的图像，不难想像出牛顿下降法迭代的过程：从初始点开始，大致向着原点不断前进。因此，如果你的迭代结果符合这个趋势，那么它很可能就是正确的。

如果取学号 $p = 2181411945$ ，程序输出的日志如下：

```
[optimizer] [debug] (138, 1620)
[optimizer] [debug] (69, 810)
[optimizer] [debug] (34.5, 405)
[optimizer] [debug] (17.25, 202.5)
[optimizer] [debug] (8.625, 101.25)
[optimizer] [debug] (4.3125, 50.625)
[optimizer] [debug] (2.15625, 25.3125)
```

```
[optimizer] [debug] (1.078125, 12.65625)
[optimizer] [debug] (0.5390625, 6.328125)
[optimizer] [debug] (0.26953125, 3.1640625)
[optimizer] [debug] (0.134765625, 1.58203125)
[optimizer] [debug] (0.0673828125, 0.791015625)
[optimizer] [debug] (0.03369140625, 0.3955078125)
[optimizer] [debug] (0.016845703125, 0.19775390625)
[optimizer] [debug] (0.0084228515625, 0.098876953125)
[optimizer] [debug] (0.00421142578125, 0.0494384765625)
[optimizer] [debug] (0.002105712890625, 0.02471923828125)
[optimizer] [debug] (0.0010528564453125, 0.012359619140625)
[optimizer] [info] 0.00015386869199573994
```


上面例子中的 logger 名字是 optimizer，你也可以换成任何其他名字（但只能包含半角英文字符）。为了在文档上看起来更美观，我们设置了 logger 的输出格式，去掉了其中的时间部分，你不必这样做。

1.4 提交内容

由于前置实验不使用 Dandelion 框架，你需要提交的内容与之后的其他实验不同。请按照附录指定的格式撰写邮件并发送到课程邮箱，附件有二：

- 将整个项目文件夹打包为 *optimizer.zip*
- 运行日志 *optimizer.log*。

你打包的 *optimizer.zip* 直接解压后应当得到一个目录 *optimizer*，该目录就是你的项目根目录。如果你打包的目录结构不符合要求，脚本将无法编译并测试你的程序，提交会被判定为无效。

 **笔记** 打包前务必删除这些文件：

- 编译结果（如 build 目录）
- IDE 缓存（如 Visual Studio 创建的 .vs 目录、VS Code 创建的 .vscode 目录，它们可能是隐藏目录）
- IDE 工程文件（如 Visual Studio 创建的 .sln 和 .vcxproj 文件、Xcode 创建的 .xcodproj 文件等）

总之，打包结果中应该只有源代码和 *CMakeLists.txt*。如果你遵循文档的要求，最终打包出的 *optimizer.zip* 应该在 10 MiB 以下，大小超过 10 MiB 的提交邮件将不会被评测。如果打包结果很大，请再次检查是否打包了无关文件。

第 2 章 配置 Dandelion 环境

配置环境（编号 1.1）是第一个正式实验，你将编译并运行 Dandelion，尝试基本的操作和功能。

2.1 实验内容

编译并运行 Dandelion，尝试加载物体、切换各种模式。

2.2 指导和要求

2.2.1 编译和运行

Dandelion 是用 CMake 管理的跨平台项目。相信经过前置实验的锻炼后，你对 CMake 已经有了基本的了解，编译 Dandelion 不会是一件难事。Dandelion 依赖的所有库已经被一并打包到 *deps* 目录下，因此你无需安装任何依赖。

具体的编译流程请参考[开发者文档：构建页面](#)。

Dandelion 会向 stdout 输出日志信息，因此你需要从终端启动（而不是直接双击可执行文件）。此后所有的文档中，我们统一使用“终端”代指 Windows 平台的 PowerShell（或 Windows Terminal）¹、Linux 或 macOS 平台的各种终端模拟器。

按照开发者文档的说明运行编译好的程序，你应该能看到 Dandelion 的图形窗口和终端输出的日志：

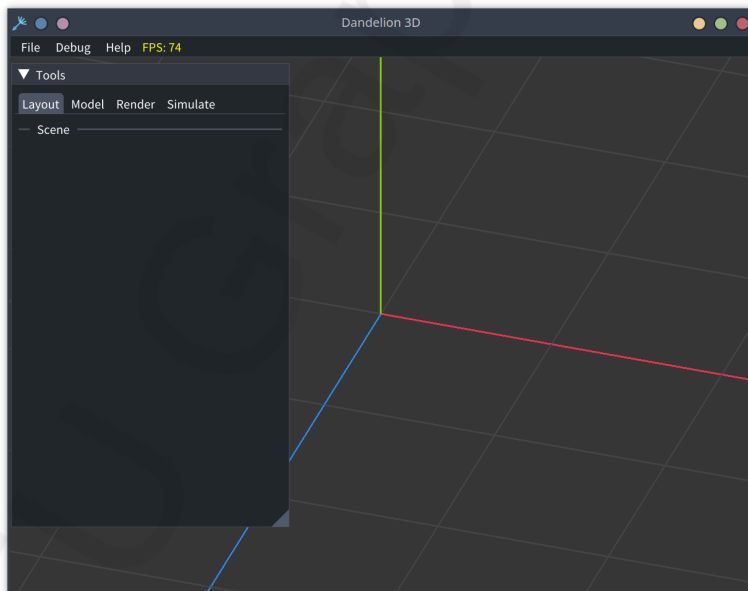


图 2.1: Dandelion 的开始界面

```
[Default] [info] Dandelion 3D, started at 2023-08-07 11:30:32+0800
[Platform] [debug] Try to create OpenGL context 4.6
[Platform] [info] runtime OpenGL context: 4.6.0 NVIDIA 515.105.01
```

¹我们非常不推荐使用 Windows 平台的同学用 cmd 作为终端。它太过老旧，对颜色、字体和编码支持都有一些问题。

```
[Platform] [info] Physical screen size: 527x296 mm, diagonal: 23.80 in
[Platform] [info] screen DPI: 123.43, scale factor: 1.5
[Platform] [info] The loaded vertex shader: resources/shaders/vertex.glsl
[Platform] [info] The loaded fragment shader: resources/shaders/fragment.glsl
[Platform] [debug] Vertex shader 1 compiled successfully.
[Platform] [debug] Fragment shader 2 compiled successfully.
[Platform] [info] Shader program 3 link succeeded
```

这就表明 Dandelion 成功运行起来了，恭喜！

图 2.1 中的界面可以分成三部分：菜单栏（顶部）、工具栏（左侧）和场景预览（整个背景）。场景中现在什么也没有，只显示了坐标轴和地平面。我们约定用红、绿、蓝三色分别代表 x, y, z 三轴的正半轴，按照 OpenGL 常用的习惯， x 和 z 轴表示水平面， y 轴表示高度，这和数学教材上的画法不太一样，但同样是右手系。地平面上有纵横交织的格子，格子的边长是 1 单位长度。刚启动时，你看到的是从 (3, 4, 5) 处观察场景的样子。

2.2.2 模式与功能

Dandelion 基本上遵循数据和计算分离的设计思想，整个三维场景的数据只有一份，通过切换不同的工作模式来使用不同的功能。它有四个模式：

- 布局模式：这是 Dandelion 启动时的模式，用于放置、移动、旋转和缩放物体。
- 建模模式：对物体进行形变和各种几何处理操作。
- 渲染模式：调整光源和相机参数，将场景渲染成图像。
- 物理模拟模式：设置物体的动力学属性，通过求解运动方程生成动画。

遗憾的是，由于现在你手中的框架还处于空缺状态，大部分功能都尚未实现。不过我们可以先尝试一些 GUI 操作——尽管很多操作现在只有数值反馈而没有视觉效果。

首先，请点击菜单栏上的 *Help -> Usage*，这时应该会弹出一个帮助窗口，包含几种常用操作的说明。

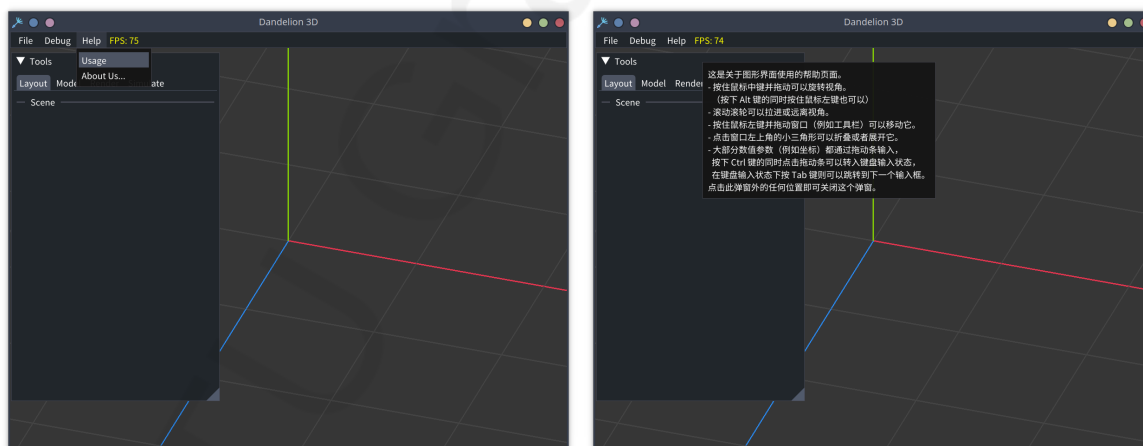


图 2.2: 点击菜单栏上的 *Usage* 项（左图）后弹出的使用帮助（右图）

你可以试试转动或者拉远视角，把工具栏挪到靠边的位置上，然后尝试下一个基本操作：加载物体。点击菜单栏的 *File -> Load File as a Group*，选择我们提供的 *cube.obj* 文件，就可以看到场景中出现了一个方块，如图 2.3 左侧所示。

一个模型文件（例如 *obj* 或者 *dae* 格式）中可能有多个 *mesh*，所以 Dandelion 将一个文件加载为一个组，将

文件中的每个 mesh 加载为一个**物体**²。组和物体的信息展示在 Scene 这一栏下，双击一个组（或单击左侧的小三角形）可以展开或折叠它，单击组中的物体就可以选中这个物体。我们选中刚才加载的方块，就会出现图 2.3 右侧所示的操纵选项。

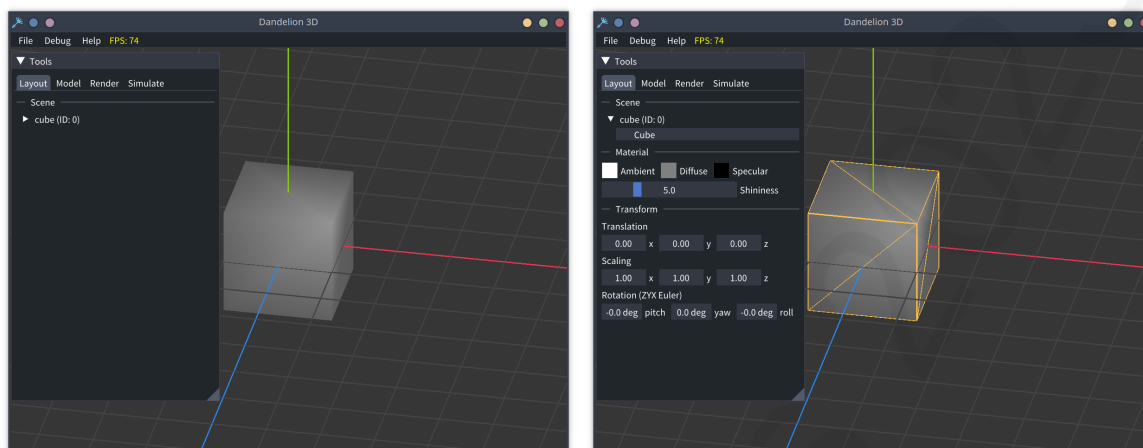


图 2.3: 加载方块（左图）和选中方块后显示的编辑操作（右图）

布局模式 图 2.3 右侧显示的就是布局模式的编辑操作，有材质编辑和（线性）变换两栏。你可以按照帮助窗口中的提示，拖动各种属性的滑动条或者直接输入数字来修改它们。材质编辑栏中是三个颜色和一个“光滑度”，具体含义留待课上解释；变换部分包含平移、缩放、旋转三项，**因为现在你还没有实现变换计算，所以只能看到数字变化，方块是不会动的**。

建模模式 在选中方块的状态下点击工具栏上的 *Model* 标签，就会切换到建模模式。此时会显示出顶点和一些箭头，如图 2.4 所示。这些箭头的含义将在几何处理部分解释，现在你只要知道看起来不一样就可以了。

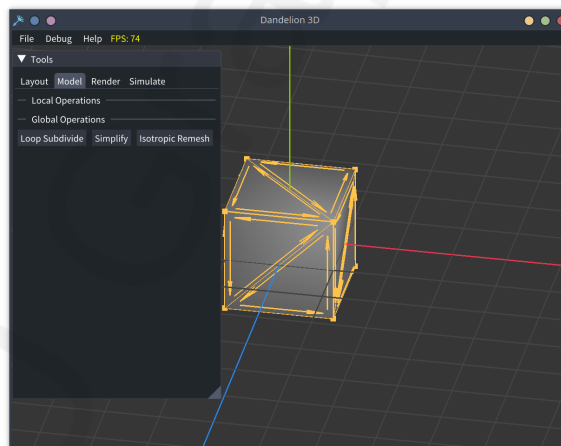


图 2.4: 建模模式下显示的物体

渲染模式 点击 *Render* 标签切换到渲染模式，此时界面上会显示出一个四棱锥形的线框，**这表示相机的视锥**，在 *Camera* 一栏下可以设置相机的各种属性，它们的含义将在渲染部分解释。点击 *Add a Light* 按钮还可以添加点光源，点光源同样可以被选中，选中后以蓝色高亮显示并可以调整它的位置和强度。

²实际上不少主流格式可以支持更加复杂的层次，但这不是我们的学习重点，因此我们只设计了简单的层次结构。

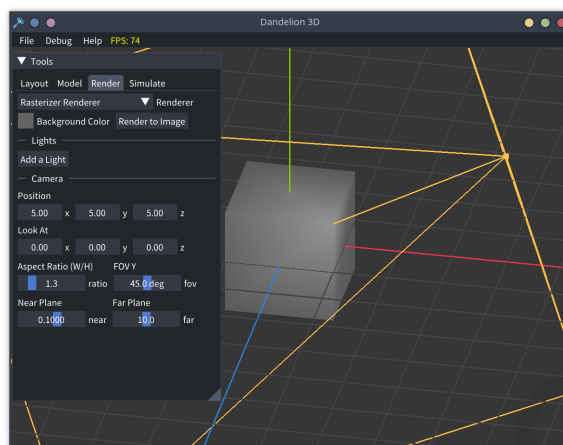


图 2.5: 渲染模式下的物体和相机视锥

物理模拟模式 点击 *Simulate* 标签切换到物理模拟模式，这个模式和布局模式有些相似，都可以看到场景中的组和物体，不同之处在于选中物体后编辑的是物体的动力学属性，包括质量、速度、合外力三项。

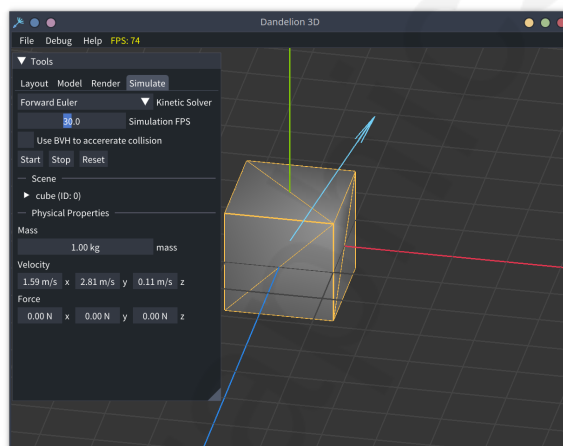


图 2.6: 物理模拟模式下会显示物体的速度

在你修改物体的速度后，屏幕上会实时显示表示速度的箭头，但由于线性变换和求解器都没有实现，现在物体是动不起来的。

你可能已经注意到了，整个场景都没有近大远小的效果。这是因为透视感依赖于透视投影矩阵，而你手中的实验框架只实现了平行投影矩阵，完成实验 1.3 后就能看到符合透视规律的画面了。

2.2.3 要求

你需要加载 *cube.obj* 文件，并在布局模式下将它的 x 坐标（也就是 Translation 属性的 x 项）设为自己的学号除以十万的结果。假如你的学号是 2181411945，你应该将坐标设为 21814.11945。由于 GUI 上只会显示两位小数，设置完成后应该看到 21814.12。

2.3 提交内容

本实验不需要编写代码，提交设置好坐标的运行截图即可。

第3章 变换矩阵

变换矩阵（编号 1.2）这个实验中，你将使用矩阵表示平移、旋转、缩放三种变换，从而构造出物体的模型变换矩阵 (Model Transformation Matrix)。

3.1 实验内容

理解三维齐次坐标和齐次坐标下的线性变换，填写平移、旋转、缩放三个变换矩阵，从而构造出物体的模型变换矩阵。完成这个实验会让工具栏 *Transform* 部分的功能生效，之后就可以在 GUI 上调整物体的位置和姿态了。

3.2 指导和要求

3.2.1 齐次坐标和变换矩阵

我们在课上已经讲到过，物体的平移、旋转、缩放等变换都可以写成线性代数的形式。在三维空间中，旋转和缩放都可以写成三维方阵左乘位置向量的形式，而平移只能写成位置向量加偏移向量的形式。为了让这些变换统一起来，我们将物体的坐标扩展一维，形成齐次坐标。一个点的齐次坐标是 $(x, y, z, 1)$ ，一个向量的齐次坐标是 $(x, y, z, 0)$ 。

在齐次坐标表示下，平移也成为了一种线性变换，于是 M_1 到 M_n 这 n 个变换可以写成矩阵连乘形式：

$$\mathbf{v}' = M_n M_{n-1} \cdots M_1 \mathbf{v}$$

这种形式可以方便地组合各种变换。

 **笔记** 每次变换都是将变换矩阵左乘到原先变换的结果上，因此从左往右读时，矩阵顺序和变换顺序是相反的。

3.2.2 平移、旋转和缩放

平移变换 平移变换用到齐次坐标的最后一维，只需要设置好矩阵最右侧的一列即可。

旋转变换 绕任意轴进行旋转变换的表达式很冗长，不太适合手写，因此我们用**欧拉角**的方法构造变换矩阵，只需要依次绕三个特殊的轴旋转即可。选取不同的旋转轴会产生不同形式的欧拉角，这个实验中我们使用 ZYX 欧拉角，依次绕模型坐标系的 z, y, x 三轴旋转。

绕自身轴（模型坐标系轴）旋转的旋转矩阵非常简单：


$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix}, R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

我们只要知道三轴的旋转角，就可以用下面的式子构造旋转矩阵：

$$R = R_z(\theta_z) R_y(\theta_y) R_x(\theta_x)$$

缩放变换 缩放变换只需要设置好矩阵的主对角线元素即可。

变换顺序 三维空间中的变换是不可交换的，如果你先进行平移再进行旋转，就会发现旋转不再是绕物体中心点进行的了。因此，我们必须将**平移变换放在最后一步**。为了方便测试，这里约定变换的顺序依次是**缩放、旋转、平移**。

 **笔记** 欧拉角一般用旋转轴次序（顺规）表示，小写字母表示世界坐标系的坐标轴，大写字母表示模型坐标系的坐标轴；也可以用加撇号的方法表示模型坐标系轴。例如我们用的 ZYX 欧拉角，也可以写作 $z-y'-x''$ 欧拉角，即先绕固定 z 轴旋转，再绕一次旋转后的自身 y 轴 (y') 旋转，最后绕二次旋转后的自身 x 轴 (x'') 旋转 [2]。

欧拉角可以分成两类：经典欧拉角 (proper Euler angles, or classic Euler angles) 的顺规中含有两个相同的轴，例如 ZXZ；而泰特-布赖恩角 (Tait-Bryan angles) 的顺规一定是三个不同的轴，例如 XYZ。我们用的 ZYX 就是一种泰特-布赖恩角。更多关于欧拉角的解释可以参考 [Euler angles - Wikipedia](#)（不要看中文版，相比于英文版缺失了太多关键内容），知乎等网站上也有很多资料。

3.2.3 要求

在 Dandelion 中，变换某个物体**不会改变它内部存储的坐标**，只是修改了它的**变换参数**（对应模型变换矩阵）。你需要填写 `Object::model` 函数，在这个函数体中根据物体的 `center`、`rotation` 和 `scaling` 三个属性计算出 model matrix。

在实现过程中需要注意：由于欧拉角存在**万向锁**等问题，Dandelion 内部是用**四元数**存储旋转参数的，你需要调用我们提供的工具函数将四元数转换成欧拉角，才能用欧拉角去构造旋转矩阵。转换方法如下：

```
const Quaternion& r = rotation;
auto [x_angle, y_angle, z_angle] = quaternion_to_ZYX_euler(r.w(), r.x(),
    ↪ r.y(), r.z());
// Then construct the rotation matrix with euler angles.
```

在正确填写 `Object::model` 函数后，在布局模式下调整工具栏 Transform 中的 Translation, Rotation 和 Scaling 就可以生效了。请加载我们提供的 `cow.dae` 模型文件，尝试调整它的平移、缩放和旋转参数。

我们提供了单元测试来验证你所写程序的正确性。如欲使用单元测试，请切换到 Dandelion 根路径下，然后按照下面的指示操作。

Windows 平台上请执行：

```
> Set-Location test
> New-Item -ItemType "directory" build
> Set-Location build
> cmake -S .. -B .
> cmake --build . --config Release --parallel 8
> ./Release/test Transformation
```

Linux / macOS 平台上请执行：

```
$ cd test
$ mkdir build
$ cd build
$ cmake -S .. -B . -DCMAKE_BUILD_TYPE=Release
```

```
$ cmake --build . --parallel 8
$ ./test Transformation
```

这个实验的单元测试将随机生成 10 组位置、缩放和旋转参数，并将你的答案与 Eigen 库提供的变换结果相比较，两个矩阵之差的 Frobenius 范数小于 10^{-2} 即判定为正确。

3.3 实验结果

你现在应该能任意调整物体的姿态了，图 3.1 是一个调整姿态的例子。

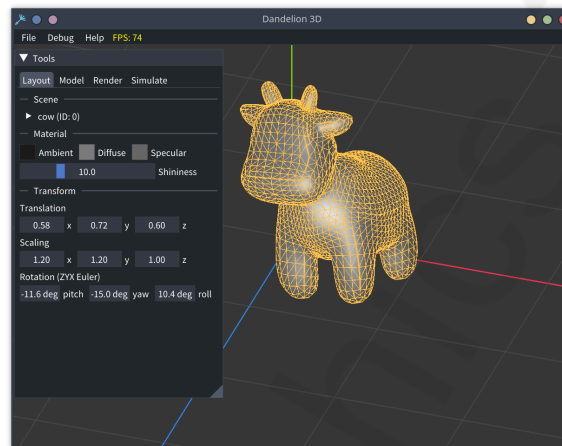


图 3.1: 调整物体姿态的例子

如果你填写了正确的变换矩阵，那么单元测试程序会输出 All tests passed :

```
[Test] [info] Dandelion 3D Unit Test, started at 2023-08-04 19:28:01+0800
Randomness seeded to: 2576752501
=====
All tests passed (10 assertions in 1 test case)
```

否则，它将输出你的程序没有通过的测试用例。

第4章 透视投影矩阵

目前为止，你看到的物体都不遵循近大远小的透视规律，这让它们的形状显得不太符合日常直觉。在透视投影矩阵（编号 1.3）这个实验中，你将构造表示透视投影变换的矩阵，从而制造透视效果。

4.1 实验内容

理解投影矩阵的作用和透视投影变换的过程，填写透视投影矩阵。完成这个实验后在界面上看到的场景将具备近大远小的特性，物体的形状将符合透视规律。

4.2 指导和要求

4.2.1 透视投影矩阵

数学上（以及绘画上）的投影指的是将某个三维区域变换到一个二维区域上的过程，也就是“一片空间”到“一张图”的过程。但图形学中的“投影矩阵”并不起这个作用，它实际上只完成投影的准备工作，将一个给定的三维区域变换到一个标准立方体区域——在我们的约定中，就是 $[-1, 1]^3$ 这个立方体区域。

对于平行投影矩阵来说，“给定的三维区域”可以是任意的长方体区域；而对透视投影矩阵，它会是一个顶面和底面平行的四棱锥，称为平截头体 (frustum)。

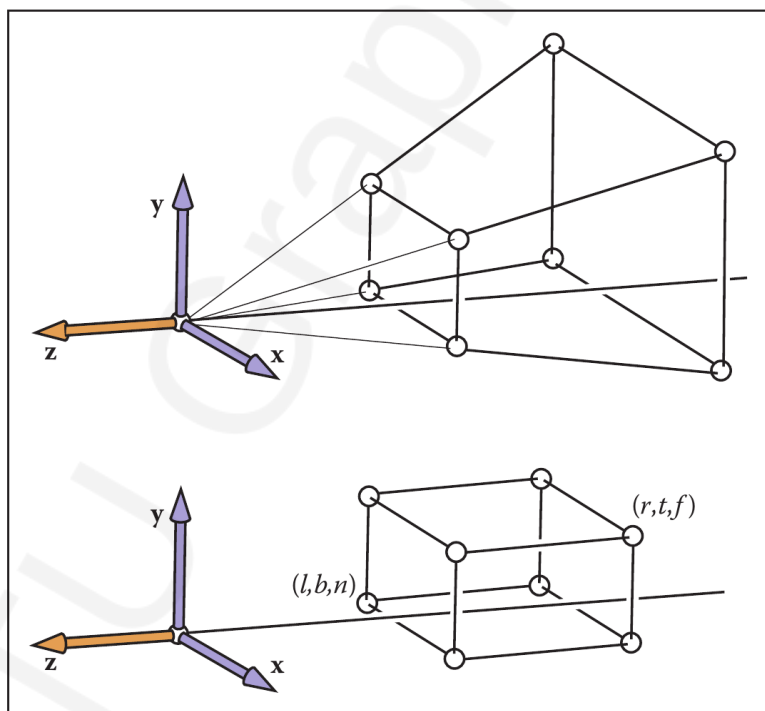


图 4.1: 透视投影矩阵的可视区域（上）和平行投影矩阵的可视区域（下） [3]

Dandelion 约定：相机（观察者）始终看向 $-z$ 方向、观察区域（可视区域）上下（左右）对称， $-z$ 方向正好穿过观察区域的中心。因此，我们只需要指定如下的参数就能确定一个平截头体：

- 视点（观察点）：相机（观察者）所在的位置。
- 目标点：观察者看向的位置。

- 视角：在观察坐标系 (view space) 下，沿着 y 方向的观察范围。
- 宽高比：平截头体任一与 z 轴垂直的剖面都是一个矩形，该矩形的宽度与高度之比 w/h 。
- 近平面和远平面：平截头体顶面与底面到视点的距离（都是正数）。

而由于投影变换在观察变换之后进行，决定透视投影矩阵的参数只有视角、宽高比、远近平面这几个。推导透视投影矩阵的方法有两种：

- 直接根据相似三角形计算变换的系数，可以参考 [OpenGL Projection Matrix](#)
- 首先将平截头体变换成一个长方体，然后进行一次平行投影变换，这是闫令琪老师在 [GAMES 101 P4](#) 课上使用的思路

两种思路都有很清晰的讲解资料，所以我们不会直接给出透视投影矩阵，请大家自己回忆课堂内容或查找资料来完成实验。

4.2.2 要求

生成投影矩阵的函数是 `Camera::projection`，关于 `Camera` 结构体和相关参数的说明请参考[开发者文档：Camera 结构体](#)。当前这个函数返回平行投影矩阵，而你需要修改这个函数，使之返回透视投影矩阵。

在计算透视投影矩阵时你会需要用到三角函数，而标准库中的三角函数是弧度制的，因此你需要用 `degrees` 函数将 `fov_y_degrees` 转换为弧度再进行计算：

```
const float fov_y = degrees(fov_y_degrees);
// Fill the perspective projection matrix...
```

我们提供了单元测试来验证你所写程序的正确性。请重新编译单元测试程序后，执行如下命令：

Windows 平台：

```
> ./Release/test "Perspective Projection"
```

Linux / macOS 平台：

```
$ ./test "Perspective Projection"
```

这个单元测试有五组预先设定的参数，将 `Camera::projection` 函数返回的矩阵与标准答案相比较，二者之差的 Frobenius 范数小于 10^{-2} 则判定为正确。

4.3 实验结果

与实验 1.2 类似，如果你的透视投影矩阵填写正确，那么单元测试程序应该输出 `All tests passed`，反之输出答案错误的用例。

此时运行 `Dandelion` 并加载 `cube.dae`，应该可以看到地平面网格和物体都遵循透视规律了。

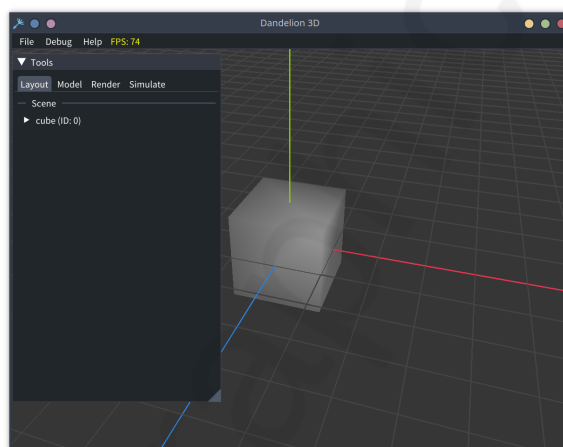


图 4.2: 透视投影下的观察结果

第 5 章 欧拉角到四元数的转换

在变换矩阵（实验 1.2）中，我们提到 Dandelion 内部是用四元数存储旋转的，在本实验（编号 2.1）中，你将实现从欧拉角转换为四元数的过程。

5.1 实验内容

作为一种表示旋转的形式，四元数比欧拉角更稳定，也不存在旋转轴选取方法繁多导致的混乱。但我们无法从四元数的一般形式 $w + xi + yj + zk$ 中直观地看出旋转是如何进行的，因此显示和操纵旋转的过程往往还是以某种欧拉角作为中间表示形式。如果选择的旋转轴都是自身轴，那么用欧拉角构造旋转矩阵也很方便。

为了在方便用户和方便计算之间取得平衡，你需要编写一个函数，将四元数转换为 ZYX 欧拉角。

5.2 指导和要求

5.2.1 欧拉角的弊端

在 3.2.2 小节中我们给出了绕三轴旋转的矩阵 $R_x(\theta_x), R_y(\theta_y), R_z(\theta_z)$ ，如果我们取 $\theta_y = 90^\circ$ ，再按 ZYX 顺序计算旋转矩阵 $R = R_z(\theta_z)R_y(\pi/2)R_x(\theta_x)$ ，会得到这样的结果：

$$R = \begin{bmatrix} 0 & \sin(\theta_x + \theta_z) & -\cos(\theta_x + \theta_z) \\ 0 & \cos(\theta_x + \theta_z) & \sin(\theta_x + \theta_z) \\ 1 & 0 & 0 \end{bmatrix}$$

这时，整个旋转矩阵 R 只由一个角 $\theta_x + \theta_z$ 决定，物体丢失了一个旋转自由度！而取 $\theta_y = -90^\circ$ 也能得到类似的结果，这就是万向锁现象：当绕某个坐标轴转动 $\pm 90^\circ$ 后，另外两个角“合并”在了一起，原本剩下的两个旋转轴现在变成了一个。这种现象在任意一种顺规下都会出现，是欧拉角的内在缺陷。由于我们采用欧拉角作为旋转的中间表示形式，你在 Dandelion 中设置航向角为 $\pm 90^\circ$ 也可以观察到万向锁现象。

另一个问题是抖动。还是以 ZYX 顺规为例：当 $\theta_y \approx \pm 90^\circ$ 时系统处于万向锁（或接近万向锁）状态，这时我们还是可以将其转换为四元数；但如果要反过来将这种姿态的四元数转换为欧拉角，在某些式子中会出现非常接近 0 的分母，进而导致物体的姿态剧烈“跳变”。在 `ui/toolbar.cpp` 的 `layout_mode` 函数中有这么两行：

```
ImGui::DragFloat("yaw", &y_angle, ANGLE_UNIT, -90.0f, 90.0f, "%.1f deg",
    ImGuiSliderFlags_AlwaysClamp);
```

你可以将这里的两个 `90.0f` 都改成 `180.0f`，重新编译之后再回到 GUI 中去操纵物体旋转，就可以在航向角 (yaw) 处于 $\pm 90^\circ$ 附近时观察到抖动了。当然，观察完后要记得改回来。

如果这两个问题都没有得到解决，那我们为什么还要使用四元数呢？首先，这两个问题是可以被四元数解决的，只不过 Dandelion 还没有处理好它们；其次，欧拉角的繁多和混乱——各种不同的旋转顺序（和转轴）在不同资料上的写法也经常不同，这导致用欧拉角表示旋转非常容易产生误会，而四元数完全没有这个问题。

5.2.2 轴-角表示法和四元数


无论选取何种顺规，欧拉角都是用三个旋转的合成来表达任意旋转的。除此以外，还有另一种表示任意旋转的方法：旋转轴和旋转角。旋转轴可以是任意过原点的直线，而物体绕旋转轴逆时针转过的角就是旋转角。显然，轴-角表示法不存在顺序选择的问题。

假如旋转轴的方向向量是 $\mathbf{k} = (k_x, k_y, k_z)^T$ ，旋转角是 θ ，我们可以用一个四元数

$$\begin{aligned} q &= (x, y, z, w) \\ &= (k_x \sin \frac{\theta}{2}, k_y \sin \frac{\theta}{2}, k_z \sin \frac{\theta}{2}, \cos \frac{\theta}{2}) \end{aligned}$$

表示这个旋转变换。

四元数用了四个参数来描述旋转，但三维空间中的旋转变换只有三个自由度，这意味着四元数的参数其实是有冗余的。另外，考虑到表示平面旋转的复数是 $\cos \theta + i \sin \theta$ ，四元数里出现 $\theta/2$ 而不是 θ 也显得有些奇怪。这都是用四维量描述三维旋转导致的，有兴趣的同学可以阅读这篇参考资料：[四元数——旋转](#)。

 **笔记** 四元数有些类似复数，但有三个虚部，通常表示为 $w + xi + yj + zk$ 或 $w + \mathbf{u}$ ， $\mathbf{u} = (x, y, z)$ ，也可以写作 $[w, \mathbf{u}]$ 。四元数的加法和复数相似，但乘法比较复杂，并且乘法不满足交换律。讲四元数运算的资料堪称车载斗量，但内容大同小异，随意选一份读即可。

四元数不是为了表示旋转而生的，单位四元数（满足 $w^2 + x^2 + y^2 + z^2 = 1$ ）才表示三维旋转变换。

用四元数表示旋转变换时有一个重要的特性：四元数的连乘可以像矩阵连乘一样表示变换的复合。所以当我们要将欧拉角转换为四元数时，也可以分别构造对应三轴旋转的四元数。不同点在于四元数的旋转轴都是固定轴（世界坐标系轴），因此我们要将 ZYX 顺规转换成等价的 xyz 顺规。转换过程很简单：绕自身 z, y, x 轴依次旋转 $\theta_z, \theta_y, \theta_x$ 角度，等价于绕世界 x, y, z 轴依次旋转 $\theta_x, \theta_y, \theta_z$ 角度。最终，我们只需要计算

$$q = q_z(\theta_z)q_y(\theta_y)q_x(\theta_x)$$

即可。 q_x, q_y, q_z 的推导留作实验任务，此处不再给出。

5.2.3 要求

你需要替换 `ui/toolbar.cpp` 中 `layout_mode` 函数的下面几行：

```
selected_object->rotation = AngleAxisf(radians(x_angle), Vector3f::UnitX()) *
                             AngleAxisf(radians(y_angle), Vector3f::UnitY()) *
                             AngleAxisf(radians(z_angle), Vector3f::UnitZ());
```

这里原先使用 Eigen 提供的 `AngleAxisf` 函数构造 q_x, q_y, q_z ，请你自己推导如何构造这三个四元数，并将构造出的三个四元数相乘得到物体的旋转变换。四元数类型由 Eigen 提供，参考 `Eigen::Quaternion` 使用 `Quaternionf` 类型即可。

5.3 实验结果

如果你正确地推导了欧拉角到四元数的转换，重新编译并运行程序时应该感觉不到任何变化——操纵物体旋转的反馈与之前完全相同，可以直接参考图 3.1。

5.4 提交和验收

本实验不需要提交截图，其他按照标准要求即可。

在验收时，你需要现场演示旋转操作并回答一些关于旋转的问题。如果任意旋转的效果依然正确，你可以得 5 分。

附录 A 如何提交实验结果

当你完成实验并检查无误后，请通过邮件提交实验结果到课程邮箱 xjtugraphics@outlook.com。由于提交实验结果的邮件数量很多，我们用脚本收集这个邮箱里的邮件并自动进行一些测试和查重。助教通常不会人工查看邮件内容，所以如果你在提交过程中遇到问题，请不要给这个邮箱发送邮件，而应该直接联系助教。

A.1 注册

在第一次提交实验结果前，你需要将自己的学号与邮箱绑定。请发送一封标题为 学号-register 的邮件来完成注册。例如你的学号是 2181411945，那么邮件标题是 2181411945-register，内容为空。绑定成功后，你将收到一封回复邮件。如果在一天后仍没有收到反馈，请尽快联系助教。

仅当你**成功注册并收到回复邮件**后，你才能开始提交实验结果。若你在收到注册成功的回复前提交实验，可能会被脚本忽略或被判定为提交错误。


A.2 邮件格式和反馈

为了确保你提交的结果能够被我们的脚本收取，请**务必保证**邮件标题为 学号-实验编号（注意学号与实验编号之间用英文减号隔开）。例如你的学号为 2181411945，要提交实验 1.3，则邮件标题应该是 2181411945-1.3，向标题中添加任何额外字符都将导致脚本无法识别你的邮件。

除了前置实验（实验 1.0）外，所有实验都使用 Dandelion 框架完成。如果没有特别说明，默认需要提交以下三项内容：

- 修改后的全部源代码：将 *Dandelion/src* 目录整个打包为 *src.zip* 文件
- 所有修改过的代码片段：将所有修改过的函数统一复制到 *modifications.cpp* 中
- 正确的运行结果截图：将符合文档要求的运行截图打包为 *screenshots.zip* 文件

提交时将需要提交的文件作为附件随邮件发送，请确保文件命名正确（建议复制这里的文件名）。

 **笔记** 打包 *src.zip* 时切勿将无关文件打包进去，正常打包出的 *src.zip* 应该不到 500 KiB 大，如果你打包出的压缩包大于 1 MiB，这个提交会被判定为无效。

整理 *modifications.cpp* 时，请完整地复制修改过的函数（整个函数定义都要复制，不要只复制自己改动的几行），否则脚本会因为找不到函数名而判定提交失败。

截图时请使用**窗口截图功能**（Windows 和 macOS 直接支持指定窗口截图，Linux 上可以使用 Spactacle 等截图工具），不要将整个桌面都截取下来。

我们的脚本每三小时检查一次邮件服务器，

- 如果它成功识别了你的提交，将给你发回一封确认邮件。确认邮件的标题一般是 学号-实验编号-提交已收到，正文部分为空。
- 如果你的提交不符合规定，脚本将发回一封提示邮件，标题为 学号-实验编号-提交错误，正文是错误原因（如果脚本不能识别则为空），请仔细检查自己的提交内容并重新提交。

运行脚本的机器停机前我们会进行公告。如果我们没有通知停机，而你在提交次日还未收到任何反馈，请先检查邮箱账户的垃圾箱；若垃圾箱也找不到反馈邮件，请及时联系助教。

A.3 实验结果自动检查

收取邮件的脚本也会自动检查提交内容，它至少会进行如下两项检查：

- 编译你的源代码，因此请勿修改 *Dandelion/src* 目录之外的任何文件、请勿在 *Dandelion/src* 目录下新增源文件，这都会导致检查不通过。
- 如果提交的实验需要查重，比对所有提交的 *modifications.cpp* 中的内容。请勿试图上传一个空的 *modifications.cpp* 或故意不将某些修改复制到其中，这种行为等同于抄袭，按作弊处理。

附录 B 选做实验

首先，恭喜你完成了所有的必做实验！接下来，让我们开始更加丰富也更具挑战性的选做实验，正式踏入图形学的大门吧。

B.1 选做实验的内容和选择方式

经典图形学有渲染、几何、动画三个主要领域，选做实验的渲染、几何处理、物理模拟三部分分别对应这三个领域中的一些基础知识。

在渲染部分，我们将介绍现代渲染管线是如何工作的，你可以实现并优化一个最简单的软光栅渲染管线。另一个主题是光线追踪，它是现代高质量渲染的基石，能够生成真实可靠的反射和阴影效果。渲染部分的实验题目中包括最简单的光线追踪算法——Whitted 风格光线追踪——来让你体验这一过程。

在几何处理部分，我们将介绍一种性质优良的辅助数据结构——半边网格，并使用这种数据结构对 mesh 进行局部或全局操作，让它更加精细或更加简单。这些几何算法虽然不能直接给出建模的结果，但足以成为建模过程中或建模完成后有力的辅助工具。

在物理模拟部分，我们将介绍如何近似计算运动和碰撞过程。一旦我们实现了模拟求解算法，只要设置好初始条件就可以让物体运动起来了，这是实现动画效果的主要手段之一。然而实时物理模拟（尤其是碰撞检测）的开销相当可观，如何加速这个过程也是一个值得思考和实践的问题。

我们不会人为限制你的选择，但某些实验之间存在自然的依赖关系：需要完成实验甲才能开始做实验乙，等等。所有的实验及其依赖关系在图 B.1 中给出。

B.2 选做实验的考核方式

图 B.1 中每个实验名字下方标注有分值，当你完成这个实验后就可以得到相应的分值。实验分数的上限为 70 分，你最终的实验总分就是选做实验与必做实验的得分之和。请注意，如果你获得的实验分数超过 70 分，核算总成绩时也只计入 70 分，即溢出的实验分数不能替代平时成绩。

与必做实验不同的是，你做完选做实验后不仅要向邮箱发送邮件来提交结果，还需要通过助教的验收才算完成实验。

大部分选做实验（图 B.1 中蓝色的那些）提交结果的方式和必做实验完全一致。每个选做实验有各自验收的内容和标准，你需要在验收时展示自己的代码、回答助教的问题并演示你的程序。我们会在学期内安排三到四次线下集中验收，每次验收都允许验收任意数量、任意部分的选做实验题目。理论上来说，你甚至可以在第一次验收时就拿到 50 分。

通常情况下，我们不会接受任何线上或单独验收的申请。当最后一次验收结束后，所有同学的实验成绩都会被冻结，此后的实验成绩只能减少而不能增加。极个别特殊情况必须提前至少一周与我们联系，我们会在慎重考虑后决定你是否可以延期验收。

B.3 代码查重与作弊处理

所有选做实验都会严格查重，如果你的代码被查出与其他同学的代码雷同，我们将作如下处理：

- 第一次查出时予以警告，查出雷同的实验即使已经验收通过也会作废，但保留一次重做这个实验的机会。
- 再次查出时直接按作弊论处，该实验成绩作废并且不再保留验收机会。

代码查重的单位是一个**实验**，例如同一天查出三个实验的代码雷同，按三次作弊处理。

B.4 挑战任务

图 B.1 中绿色的实验是“挑战任务”，它们综合性更强、难度更大并且**没有已经成型的代码框架和步骤指导**。这些任务是我们从 Dandelion 框架目前尚未实现的功能中选出的，面向对图形学感兴趣的同学开放。如果你完成的结果质量较高，你的成果将最终合并进入 Dandelion 框架。

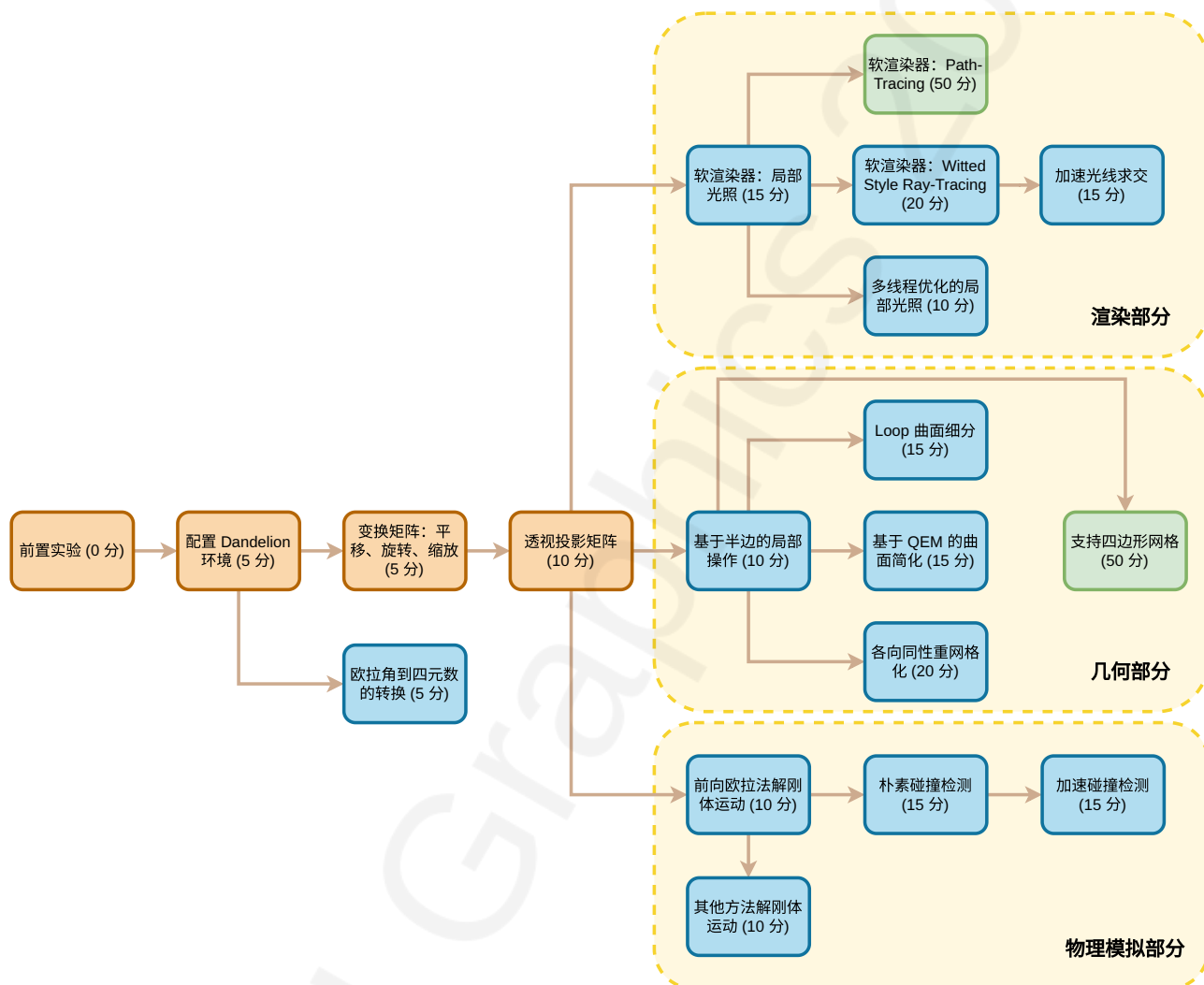


图 B.1: 所有实验及其依赖关系

参考文献

- [1] Ag2gaeh. *Parabol-el-zy-hy-s*. Wikimedia Commons. published under CC BY-SA 4.0 license. 2017. URL: <https://commons.wikimedia.org/wiki/File:Parabol-el-zy-hy-s.svg>.
- [2] *Euler angles*. Wikipedia. URL: https://en.wikipedia.org/wiki/Euler_angles.
- [3] Steve Marschner and Peter Shirley. *Fundamentals of Computer Graphics, Fourth Edition*. CRC Press, 2015.